Policy-Based Runtime Verification of Information Flow

PhD Thesis

Mohamed Khalefa Sarrab

This thesis is submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Software Technology Research Laboratory

Faculty of Technology

De Montfort University

March 2011

DEDICATION

To my beloved parents

The thesis is dedicated to my loving father, **Mr. Khalefa Mohamed Sarrab**, who has been a great source of motivation, inspiration and endless support throughout my life, and who sacrificed a lot for me to be what I am now.
It is also dedicated to my loving mother, **Mrs. Ramadana Said Sarrab** who gave her love and support, for everything she sacrificed in her life for me. Without her loving care, prayers and support, it would have been very difficult for me to achieve my objectives.

To my beloved family

I would like to dedicate this thesis to my beloved wife **Mrs. Laila Albogdadi Elgamel** and to my children **Rahaf** and **Khalefa**.

I owe everything I have achieved or will achieve to them. I hope that by obtaining my PhD I can put smiles on their faces.

ABSTRACT

Standard security mechanism such as Access control, Firewall and Encryption only focus on controlling the release of information but no limitations are placed on controlling the propagation of that confidential information. The principle problem of controlling sensitive information confidentiality starts after access is granted. The research described in this thesis belongs to the constructive research field where the constructive refers to knowledge contributions being developed as a new framework, theory, model or algorithm. The methodology of the proposed approach is made up of eight work packages. One addresses the research background and the research project requirements. Six are scientific research work packages. The last work package concentrates on the thesis writing up.

There is currently no monitoring mechanism for controlling information flow during runtime that support behaviour configurability and User interaction. Configurability is an important requirement because what is considered to be secure today can be insecure tomorrow. The interaction with users is very important in flexible and reliable security monitoring mechanism because different users may have different security requirements. The interaction with monitoring mechanism enables the user to change program behaviors or modify the way that information flows while the program is executing. One of the motivation for this research is the information flow policy in the hand of the end user.

The main objective of this research is to develop a usable security mechanism for controlling information flow within a software application during runtime. Usable security refers to enabling users to manage their systems security without defining elaborate security rules before starting the application. Our aim is to provide usable security that enables users to manage their systems' security without defining elaborate security rules before starting the application. Security will be achieved by an interactive process in which our framework will query the user for security requirements for specific pieces of information that are made available to the software and then continue to enforce these requirements on the application using a novel runtime verification technique for tracing information flow.

The main achievement of this research is a usable security mechanism for controlling information flow within a software application during runtime. Security will be achieved by an interactive process to enforce user requirements on the application using runtime verification technique for tracing information flow. The contributions are as following.

- **Runtime Monitoring**: The proposed runtime monitoring mechanism ensures that the program execution are contains only legal flows that are defined in the information flow policy or approved by the user.
- **Runtime Management**: The behaviour of a program that about to leak confidential information will be altered by the monitor according to the user decision.
- **User interaction control**: The achieved user interaction with the monitoring mechanism during runtime enable users to change the program behaviors while the program is executing.

DECLARATION

I declare that the work described in my thesis is original work undertaken by me for the degree of Doctor of Philosophy, at the Software Technology Research Laboratory (STRL), De Montfort University, United Kingdom. No part of the material described in this thesis has been submitted for the award of any other degree or qualification in this or any other university or college of advanced education.

Mohamed Khalefa Sarrab

ACKNOWLEDGEMENTS

In the name of **Allah**, the Most Merciful and the Most Gracious, I give praise and thanks to Him for supporting me with the strength to complete this research, and for providing me with knowledgeable and caring individuals during the study process. Without Him, none of this work would have been possible. This thesis could not have been completed without the recommendations, advice and suggestions of many people. It may not be possible to mention all of them here, but their contributions, guidance and support are extremely appreciated.

Studying at the University of De Montfort in Leicester and particularly at the STRL was the most rewarding experience I have ever had. To me, this is certainly related to the high standard of the facilities offered to students, the friendly atmosphere among the staff and the research students, and above all the excellence of supervision.

For this reason I would like first to express my deepest appreciation and lasting gratitude to my first supervisor **Dr. Helge Janicke**, his wide knowledge and his logical way of thinking have been of great value to me. His understanding, encouragement and personal guidance have provided a good basis for the present thesis. Without his guidance the successful completion of the research and this thesis might have been a very difficult task. His critique and helpful ideas showed me the way to proceed. I am really grateful for that. I really appreciate his positive comments for improving and bringing out the optimum consequences from my endeavors. Without

his guidance and support I would never have been able to organize and complete my task well and within time constraints. The only thing I can tell him is *thank you*.
I would also like to express my thanks and appreciation to **Dr.Antonio Cau** my second supervisor and **Prof. Hussein Zedan**, the head of the STRL, for their valuable comments, constructive criticism, scientific support, insightful comments, guidance and suggestions, without which this thesis could not have been produced in the present form.
I wish to thank all researchers, colleagues and staff of the STRL. During this work I have collaborated with many colleagues for whom I have great regard, and I wish to extend my warmest thanks to all those who have helped me with my work in STRL. I want to thank them for all their help, support and valuable hints. I enjoyed these, as they gave me the feeling of belonging to a group. My greatest debt is to my father **Khalefa Sarrab**, my mother **Ramadana Sarrab**, my wife **Laila Elgamel**, my brothers (**Ali** and **Aymen**), My sisters (**Hanan**, **Samah** and **Asma**), My children (**Rahaf** and **Khalefa**) and my uncle, who is my second father **Abdullah Sarrab**. The list of sacrifices they have made in order to complete my study is endless.
Finally, here at Leicester and especially at De Montfort, I have had a great time and an enjoyable experience. I would like to thank all the good friends whom I have met in Leicester. First, there are five kind people whom I have met in Leicester: **Abdulhamed Elaskri**, **Mahmoud Elbasir**, **Abdulbasat Ghndi**, **Shaban Al-Rmalli** and **Abdulgader Grain**. Special thanks are also due to my closest friends and my officemate, **Dr.Nasser Al-Alwan**, for his support and contributions. In this city I have met all of you, so our friendship started here and will continue forever.

Thank you.

Leicester, England, 2011

Mohamed Sarrab

PUBLICATIONS

1. **M. Sarrab. H. Janicke and A. Cau.** ***Interactive runtime monitoring of information flow policies.*** **In proceedings of Second international conference of Creativity and Innovation in Software Engineering, 2009.**

2. **M. Sarrab and H. Janicke.*Monitoring explicit information flow using Java byte-code instrumentation.* In proceedings of the 8th International Conference On Computer Applications (ICCA), Yangon, Myanmar March, 2010.**

3. **M. Sarrab and H. Janicke.** ***Runtime Monitoring of Information Flow Policies.*** **In proceedings Doctoral/PhD Workshop of 4th ACM International Conference on Distributed Event-Based Systems (DEBS-2010), 11 July 2010, Cambridge, UK, 2010.**

4. **M. Sarrab and H. Janicke.** ***Runtime Monitoring and controlling of Information flow*** **. International Journal of Computer Science and Information Security (IJCSIS) Vol. 8, No. 9. pp 37-45. December, 2010.**

Contents

List of Figures

Listings

List of Tables

List of Acronyms

JVM Java Virtual Machine

SPI Security Pipeline Interface

MAC Mandatory Access Control

DAC Discretionary Access Control

NDAC Non-Discretionary Access Control

IFS Information Flow Stack

IMFS Implicit Information Flow Stack

BLP Bell LaPadula Security Model

Chapter 1

Introduction

Objectives

- Motivate the needs of information flow control.
- Highlight the original contribution and identify the research question.
- Provide the research methodology and define the success criteria.
- Provide the thesis organization.

1.1 Introduction

As our businesses, government and military become increasingly dependent on modern information technology, computer application security protection against malicious code and software system bugs become increasingly important. The more sensitive the information, such as credit card data, government intelligence, military or personal medical information being processed by software, the more important it is to ensure information confidentiality. The leakage of confidential information may cause financial damage in case of loss or destroy private or sensitive secret information. As an example Trusted Solaris Sun Microsystems (2000) uses a security technique that determines which information is accessible by users, using a mandatory access control mechanism.

However, in many cases discretionary access mechanisms that are usable, reliable and can protect the confidentiality and integrity of sensitive information accessed by any untrusted software are more suitable as they do not involve the source level of administration and grant users discretion about how their information is being used. Information flow occurs from a source (subject) to a target or destination (object) whenever information stored in a source is propagated directly or indirectly to a target object. An example flow would be the copying of a file into an email that is subsequently sent through the network. The following informal example illustrate this.

Assuming that some sensitive information is stored on a computer system, how can we prevent it from being leaked? The first approach that comes to mind is to limit access to this sensitive information, by using any type of access control mechanisms or using encryption or firewalls mechanisms. These are very useful approaches which, however, have their limitations. Standard security mechanisms are focused only on controlling the release of information but no restrictions are placed on the

propagation of that information and thus are unsatisfactory for protecting confidential information. For this reason, the proposed approach of this thesis is controlling the flow of the information from source to destination during runtime based on Java bytecode instrumentation.

1.2 Scope of the Thesis

Standard security mechanisms such as Access control, Firewall and Encryption (Anderson 2001, Bishop 2003) only focus on controlling the release of information but no limitations are placed on controlling the propagation of that confidential information. The approach controls the flow of the information only within one application. The scope of the thesis includes in particular.

- **Configurable information flow** Configurability is an important requirement because what is considered to be secure today can be insecure tomorrow. A property of configurable information flow is that it provides flexible security mechanisms that can control changeable security requirements.

- **User interaction control** Interaction with users is very important in flexible and reliable security monitoring mechanism because different users may have different security requirements. These cannot always be anticipated prior to the execution of the program. Users interact with a monitoring mechanism during runtime, enabling them to change program behaviours or modify the way that information flows.

This research is part of a wider research project that addresses information flow and dissemination control in a wider information system context. The focus is on a single Java application that represents untrusted code. This is particularly relevant in the light of increasing smart phone applications that are freely available to a very

large number of users. End users have little or no assurance that the execution of there (*apps*) does not communicate their personal information to untrusted parties.

1.3 Research Question

The overall and the central research question investigated in this thesis is.

How to control information flow from source to destination during the runtime of an application and with support of user interaction.

In order to answer the central research question, a set of common issues has been defined to address the problems of information flow control in detail:

- Dynamic information flow policy.
- User ability to modify the flow policy during runtime in response to incidents.
- Changing the programs behaviour that is leaking confidential information according to the user decision.

1.4 Original Contribution

The main contribution of this research is to develop a usable security mechanism for controlling information flow within a software application during runtime. Usable security refers to enabling users to manage their systems security without defining elaborate security rules before starting the application. Security will be achieved by an interactive process in which our framework will query the user for security requirements for specific information that are made available to the software and then continue to enforce these requirements on the application using runtime verification technique for tracing information flow. The original contributions of the thesis are as follows:

- **Runtime Monitoring**: The monitoring mechanism ensures that the program contains only legal flows those are defined in the information flow policy or approved by the user. Traditional runtime monitoring are not suitable for monitoring information flow or managing the program behaviour at runtime, as there is no feedback from the monitor to the observed software.

- **Runtime Management**: The behaviour of a program leaking confidential information will be altered by the monitor according to the user decision. Analysing the impact of a user or policy induced program alteration with the program original functional requirements is an open question.

- **User interaction control**: The achieved user interaction with the monitoring mechanism during runtime enable users to change program behaviours or modify the way that information flows while the program is executing. To our knowledge these have not been done before.

1.5 Research Methodology

The research method used in this approach is a typical scientific research technique (Wilson 1991). As in the majority of the computer science approaches the described research belongs to the constructive research field where the constructive refers to knowledge contributions being developed as a new framework, theory, model or algorithm.

The methodology of the proposed approach is made up of eight work packages. One addresses the research background and the research project requirements. Six are scientific research work packages. The last work package concentrates on the thesis writing up. The investigation work packages are illustrated in Figure 1.1.

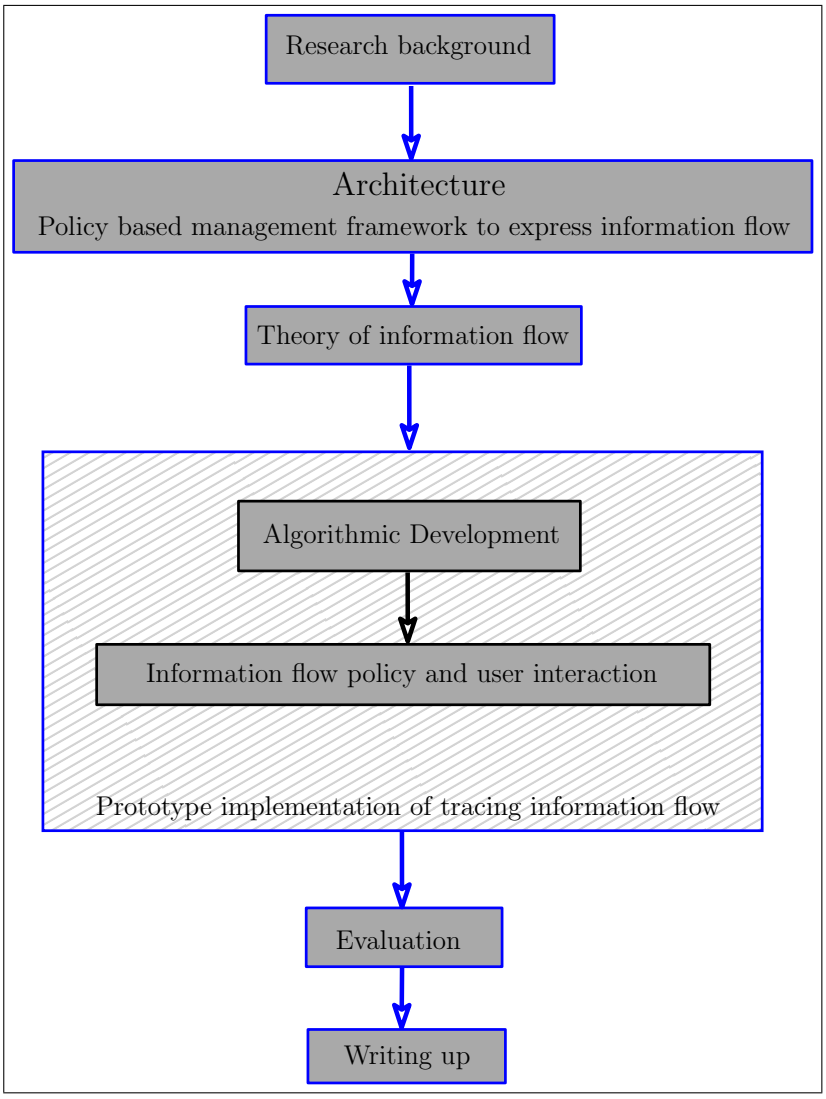


Figure 1.1: Research work packages

- **Work package 1:** The research background.

  The research background will start with a theoretical literature review including understanding of all approaches related to the research question. To achieve the objective of this step, digital resources such as the Google search engine, IEEE Xplore, SpringerLink, ACM Digital Library and CiteSeer are going to be used.

- **Work package 2:** Architecture.

  This work package will focus on the design of the framework architecture to

capture the research objectives as expressed in the research question. This work package will specify all components of the proposed framework. The research in this work package explicitly states how the framework components interact to achieve the research objectives. In this work package the work is split into two tasks.

1. Policy model for information flow control.
2. User interaction.

- **Work package 3:** Theory of information flow.

  The research investigation in this stage will focus on the development of a novel theory for controlling information flow which supports user interaction during runtime. This work package is split into two tasks.

  1. Direct information flow control.
  2. Indirect information flow control.

- **Work package 4:** Algorithmic Development.

  This research work package investigates each of the Java program phases before execution such as loading, linking and initialization. The main part of this work package is focused on providing a new instrumentation algorithm to monitor the program behaviour and to provide a flexible instrumentation mechanism that is applicable to Java bytecode. In this work package the research will concentrate on describing the architecture used to validate information flow requirements expressed as policies using runtime verification. This work package focus on the Java bytecode instrumentation process to monitor and control the target program behaviour with the respect of the information flow policy. This work package is split into two tasks.

  1. Loading of Java bytecode.

2. Java bytecode instrumentation algorithms.

- **Work package 5:** Information flow policy and user interaction.

  The investigation in this work package will concentrate on how to monitor and control the flow of the information with respect to an information flow policy. The main objective is to show how the user interacts with the monitoring mechanism, the user ability to change the program behaviour and modifying the information flow policy during runtime. The research in this work package is split into two tasks.

  1. Development of an information flow policy.
  2. User feed back.

- **Work package 6:** Prototype implementation of tracing information flow.

  This work package of the research will describe the design and implementation of our prototype for tracing the information flow which depends on the completion of work package 5.

- **Work package 7:** Evaluation.

  After building the architecture of the runtime monitoring mechanism the effectiveness evaluation of the proposed approach will take place using small but representative case studies.

  This work package is split into two tasks.

  1. A small scale case study that shows how a Java program will be traced.
  2. A medium scale case study of file sharing system showing how the information flow will be controlled.

  This work package demonstrates the practical applicability of the presented research. A conclusion was provided from the experiences of the evaluation

phase. A number of potential extensions for this research study was raised to motivate further investigation in the field of information flow.

- **Work package 8:** Write up.

  Writing up of the thesis which is based on the results of all work packages.

1.6 Success Criteria

The measure of success is that both the framework model and their supporting algorithm indeed resolve the proposed research question and demonstrate it by experiments through the implementation prototype. The prototype demonstrates that the research output results match the research objectives as follows:

- User ability to modify the flow policy during runtime in response to incidents.
- Modifying the behaviour of the program that is leaking confidential information according to the user decision.
- With reasonable performance. Some experiments will be used to measure the performance overhead in both computation time and memory usage.

1.7 Thesis Organisation

The previous sections have provided an overview of the thesis scope, original contributions and research question of this thesis. The research was undertaken along a theoretical to applied axis and was structured in work packages and was transformed into thesis chapters in the writing up stage. The rest of the thesis is organised in chapters as follows.

- **Chapter 2** introduces basic concepts of security and information flow. This chapter provides an introduction to the principles of security, static informa-

tion flow analysis and dynamic information flow analysis. It gives an overview of existing approaches of bytecode instrumentation and information flow control and discusses the difficulties and problems of the related research. This chapter provides an introduction and critically reviews related work in the areas of access control model and take grant model; Static information flow analysis, dynamic information flow analysis, bytecode instrumentation, information flow control, information flow analysis, program slicing and program dependences techniques.

- **Chapter 3** provides a general overview of the proposed framework and describes the framework architecture. In this chapter the focus is on how components of the proposed framework interact to trace and control the information flow within a Java application.

- **Chapter 4** provides an introduction to Java bytecode, class loader in the Java virtual machine and how Java class file will be instrumented in order to monitor and control information flow within a Java application. The focus in this chapter is on loading and instrumenting a target program. This chapter also shows how Java bytecode instructions will be instrumented.

- **Chapter 5** describes the second step of our runtime monitoring mechanism: how a class file will be executed and monitored to control information flow based on the information flow policy. This chapter describes the event recognizer and runtime checker algorithms for controlling information flow within a Java application.

- **Chapter 6** discusses the information flow policy and user feedback component. The chapter provides a general overview of information flow requirements and describes the information flow policy language. This chapter also

focused on the user interaction with the runtime monitoring mechanism during runtime to change the program behavior and modify the information flow policy.

- **Chapter 7** provides an introduction to high level design of the developed prototype for controlling information flow. It also gives a brief introduction to the runtime monitoring mechanism components that are used in the prototype and how they interact to load, instrument and control the flow of information in the target class files.

- **Chapter 8** provides two case studies to illustrate the practical applicability of the presented research. The first case study is provided to demonstrate the work of instrumentation process, event recognizer and the runtime checker. The second case study is presented to show how a Java class file will be traced and monitored.

- **Chapter 9** evaluates the research which has been described in this thesis and discusses the limitations of the proposed approach.

- **Chapter 10** summarises the research and proposes future work.

Chapter 2

Background and Related Research

Objectives

- Provide an introduction to the security principles, static information flow analysis and dynamic information flow analysis.
- Give an overview of existing approaches.
- Identify the difficulties and problems of the related research.

2.1 Introduction

This chapter provides an introduction to the principles of security, static information flow analysis and dynamic information flow analysis. It gives an overview of existing approaches of bytecode instrumentation and information flow control and discusses the difficulties and problems of the related research. This chapter is divided into nine sections. Section 2.2 provides an introduction to security and a critical review of related work in the areas of access control model and take grant model. Section 2.3 introduces the notion of information flow control and highlights its different types. Section 2.4 provides an introduction to static information flow analysis and a critical review of related research in the areas of type system approaches and semantic approaches. Section 2.5 discusses dynamic information flow analysis and gives a critical review of related approaches in the areas of dynamic analysis at binary code level, source code level and event level. Section 2.6 provides different mechanisms of Java bytecode instrumentation. Section 2.7 critically reviews the related research in the area of information flow control including explicit information flow control and implicit information flow control. Section 2.8 presents some examples of existing implementations of information flow analysis (JFlow, Flow Caml and Bytecode verifier). Section 2.9 discusses different types of program slicing including forward slicing, backward slicing, static slicing and dynamic slicing. Finally 2.10 presents program dependencies techniques including control dependencies and data dependences.

2.2 Security

Security in information flow control can be defined as the prevention of sensitive information to leak. Security in the context of information flow has the following

three components.

- **Requirements** Information flow requirements should define the information flow goals. They should provide the answer the question.

  How the information should flow in the system?

- **Information flow policy** The information flow policy defines the meaning of secure information flow. It provides the answer to the question.

What steps should be taken to detect and prevent the leak of the information?

- **Information flow mechanisms** The information flow mechanism enforces the information flow policy. That should provide the answer to the question.

What tools and methods are used to ensure that the previous steps are followed?

2.2.1 Access Control Model

Access control mechanism is the first technique that has been developed and it is widely deployed because of the assumptions that it enforces confidentially. This assumption comes from the fact that the subject *user and process* can not leak confidential information about any object without having access to that object. Therefore, the main idea behind access control techniques is to place access restriction on processes to limited the access to a number of objects. There exist various access control models.

Discretionary Access Control (DAC) is the most used way of access control. In this type of access the access rights of every subject on every object *file* should be

explicitly stated by the owner of that object in a policy. The owner should also state what privileges the subjects have on that object as in *UNIX* discretionary access control used for data protection in a file system, where the owner of the object can modify the access policy to control the access to it.

Mandatory Access Control (MAC) in this type of access control the policy is under the control of an administrator not under the control of the owner. Each subject and object in mandatory access control has an associated security level *security clearance*. The most popular example of mandatory access control is Bell-LaPadula model (LaPadula & Bell 1973). In this model the decision of accessing an object is taken by comparing the subject security level with the target or object security level and the security levels form a lattice. Therefore, the subject can (access, read, write, execute) object only if the subject security level is greater than or equal to the object security level. In mandatory access control the Trusted Computing Base (TCB) (H.Saltzer & D.Schroeder 1975) is usually used to compute the security level of newly created data. Another example of MAC is the Chinese Wall Security Policy (Brewer & Nash 1989).

Non-Discretionary Access Control (NDAC) is the third common access control mechanism which is based on the subject's role or tasks assigned to the subject to allow or reject object access. Non-discretionary access control is also called task-based access control or role-based access control. It works well when security based on roles or tasks is needed. As mentioned by Bandara (Bandara et al. 2007) "authority is vested in some users, but there are explicit controls on delegation and propagation of authority".

Of course these techniques are important and very useful for limiting the access of data. However, these techniques are insufficient for protecting the confidentiality of the information because, once the access is granted there is not any control the propagation of that information.

2.2.2 Take Grant Model

Another important security model that should be mentioned is provided by the Lipton and Snyder (1977) *Take Grant Protection Model*. It is a formal model provided to disprove or establish safety properties in a given system. This model uses directed graph to present the subject access to an object. The nodes in the graph represent subjects and objects. The edges between the subject and object are labeled and indicate the access rights that the subject of the edge has over an object. The two fundamental access rights in this model are *take and grant* rules.

- **Take** allows any subject *s1* in the graph to take the rights of any other subject *s2*.

- **Grant** allows any subject *s1* in the graph to grant its own rights to any other subject *s2*.

This model of dynamically rewriting rules in the graph describes acceptable changes in the graph, e.g. adding new entities or creating new edges with respect to take and grant rights (Janicke 2007).

2.3 Information flow control

Information flow control aims to fill the gaps left by standard security mechanism (Anderson 2001, Bishop 2003) by considering the flow of the information within a system when enforcing an information flow security policy. Information flow occurs from source objects to target objects, whenever information is read from a source it is potentially propagated to the target object. There are two types of information flow direct information flow and indirect information flow (Genaim & Spoto 2005, Denning & Denning 1977).

2.3.1 Direct Information Flow

Direct information flow is defined as the operation that generates a flow between a source and a target independent of any other objects. Direct information flow can be explicit or implicit.

Direct explicit information flow

Direct explicit information flow is a data flow that arises at for example assignment statements Listing 2.1 is an example of direct explicit information flow.

Listing 2.1: Example of direct explicit information flow

```
Dest := Source;
```

For statement *Dest = Source* in Listing 2.1 there is direct explicit flow *Source* $\longrightarrow$ *Dest.* i.e. a value is directly passing from the *Source* object to the *Dest* object (see Figure 2.1).

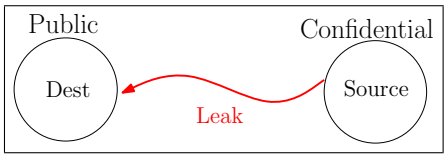


Figure 2.1: Directly passing data value from Source to Destination

Direct implicit information flow

Direct implicit information flow is a data flow that arises from for example a conditional statement, see Listing 2.2.

Listing 2.2: Example of direct implicit information flow

```
low:=0;
if high==1 then
    low:=2
 else
    skip;
```

For statement *if High==1 then* in Listing 2.2 there is a direct implicit information

flow *High* $\longrightarrow$ *Low* since changes of the values of the *High* object are observable from the values of the *Low* object.

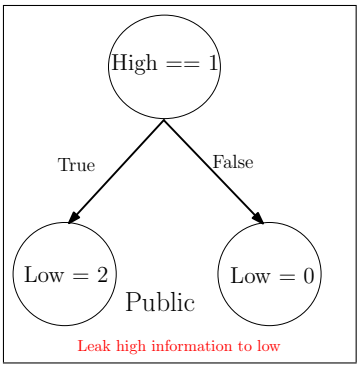


Figure 2.2: High process updates a low variable.

2.3.2 Indirect Information Flow

Indirect information flow means that there is an operation generating a flow from a source to destination and the operation is dependent on the value of other objects. Indirect information flow, also called transitive flow arises for example as a composition of direct information flow, see Listing 2.3.

Listing 2.3: Example of indirect information flow

```
x = y + z;,
w = x;
```

For statement **x = y + z;** in Listing 2.3 there is a direct information flow from y and z to x (y $\longrightarrow$ x , z $\longrightarrow$ x) and in the second statement *w = x;* there is direct information flow from x to w (x $\longrightarrow$ w) which leads to an indirect information flow from y to w and z to w (y $\longrightarrow$ w and z $\longrightarrow$ w), see Listing 2.3. Hence, direct information flow does not require any mediation between the objects to exchange, read, write or execute information. In contrast indirect information flow always requires mediation between two objects (Zhang & Yang 2002, Herrmann 2001).

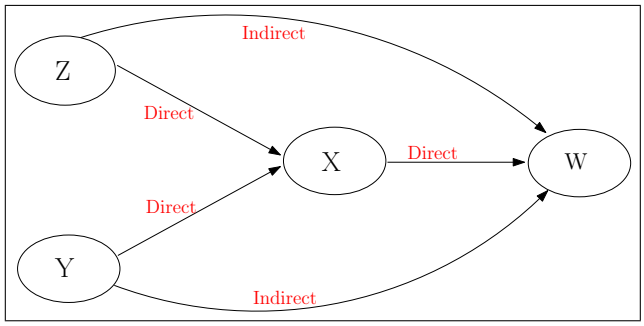


Figure 2.3: High process updates a low variable.

2.4 Static Information Flow Analysis

The static verification involves the analysis of source text by humans or software which can help to discover errors early in the software process (Havelund & Goldberg 2005). Security requirements in information systems change more frequently than functional requirements especially when new users or new data is added to the system. Runtime verification (Janicke et al. 2005, Kim et al. 1999, Lee et al. 1998) has been used to increase the confidence that the system implementation is correct by making sure it conforms to its specification at runtime. Static information flow analysis is a form of information flow analysis that does not require the system to be executed or operated. The majority of information flow analyses approaches are based on static information flow analysis, that attempt to analyse how information flows in the software to determine whether it obeys some predefined policy with respect to an information flow without running the program (Banerjee & Naumann 2005, Myers 1999). Software inspection is one of the most important form and widely used techniques for static analysis in the earlier stages of the software development. Software inspection is concerned with detecting software defects such as Fagan inspection (Fagan 1986, *Michael Fagan Associates* 2010) that is a process focused on detecting faults in the software development life cycles. NASA also has another process that inspect software statically called *Software Formal Inspection*

Process Standard (NASA 1993, SATC 2002).

The research domain in the field of static analysis for information flow has been influenced by the proposed work of Goguen and Meseguer (1982) where they define the notion of non-interference as handling the occurrence of *illegitimate* information flow in a system specification. Suppose that two security level of confidentiality *high* and *low* exist and that *highly* secret data should never flow to *low* level subjects. Non-interference declares that information cannot flow from *high* to *low* whenever *high* level cannot interfere with *low* level. It can also be defined as that the process is said to be non interfering if its *low* outputs do not depend on it is *higher* inputs. Sabelfeld and Myers (2003) in their recent survey present about 147 research references on information flow security. The vast majority of these publications are concerned with describing and defining the notion of Non-interference, e.g. (P.Allen 1991, Bieber & Cuppens 1992, McCullough 1988, Sutherland 1986, Wittbold & M.Johnson 1990). Most of these approaches are based on Goguen and Meseguer (1982) approach for non-interference, the idea of non-interference is based on, how to characterise the absence of any flow that resides at a more abstract level than the security access control models and providing it as a formal semantics to the one path flow intuition behind terms like read and write. The most important categories of static analysis approaches are type system approaches and semantic approaches.

2.4.1 Type System Approaches

Security type system can be expressed as a set of typing rules in the form of judgments that can used to describe which security level is allocated to a program (or expression) based on security levels of sub-programs or sub-expressions. The idea of security type system comes from a lattice $L, \leq$, where L is a set of security levels, e.g. *Top-secret, secret, confidential and unclassified* including a lowest or bottom element $\bot$ and highest or top element $\top$. The most earlier and useful work related

to certification of secure information flow is done by Denning et al. (1977). They provided a static certification mechanism that verifies secure information flow in the program. Their certification method is essentially a type based approach based on a lattice structure of security classes to validating secure information flow. In their method each object is assigned a security class with respect to a lattice structure. A partial order relation is a class of binary relations with the following characteristics.

1. Reflexivity.

$$a \leq a \textit{ if the relation} \leq \textit{returns true for the input (a,a).}$$

2. Anti symmetry.

$$\textit{if } a \leq b \textit{ and } b \leq a \textit{ then } a \textit{ equal } b.$$

3. Transitivity.

$$\textit{if } a \leq b \textit{ and } b \leq c \textit{ then } a \leq c.$$

The relation $\leq$ is a partial order relation if it satisfies all previous properties. A partially ordered set is a set together with a partial order relation and is called a poset. A lattice is a poset in which all non-empty finite subsets have both a least upper bound and a greatest lower bound. Denning (1976) describes a lattice of subset of $W = \{a,\ b\ ,c\}$ as follows.

$$\textit{Security class} = \textit{powerset(W)}$$

$$\mathrm{a} \longrightarrow \mathrm{b} \text{ iff } \mathrm{a} \subseteq \mathrm{b}$$

$$\mathrm{a} \oplus \mathrm{b} = \mathrm{a} \cup \mathrm{b}$$

$$\mathrm{a} \otimes \mathrm{b} = \mathrm{a} \cap \mathrm{b}$$

$$\mathrm{L}{=}\phi \text{ , H=W}$$

Description

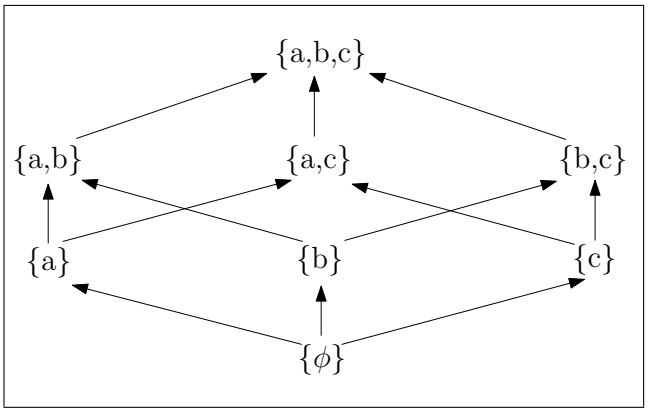


Figure 2.4: Lattice representation

In their certification method (Denning & Denning 1977) the security classes are declared in the declaration part of the program with respect to a lattice structure e.g.

Listing 2.4: Declaration of security classes

```
i,j:integer security class L;
a,b:boolean security class L;
x : file security class L;
h :integer security class H;
y,z:file security class H;
```

Their approach is sufficiently simple so that it can be used in the analysis phase of any compiler. They tried to prove that a program can not cause public output that depend on secret input. However, their method certifies only secure programs because it doesn't distinguish between secure and insecure executions of the same program. Thus, the whole program will be rejected as insecure because the set of all possible paths of the program execution must be secure. Andrews and Reitman (1980), Denning and Denning (1977) build their argument for secure information flow on the intuition that a secure information flow can be produced from a combination of secure information flows. However, both of them never formally prove that, the rules of secure information flow in type system approach are used to verify

weather a typing environment y is compatible with a given software. A typing environment can be mapped to a security class according to variables identified with respect to the lattice structure *(L, ≤)*. The rules of a sound type system approach for secure information flow as provided by Volpano, Smith and Irvine (1996) is that if a typing environment (y) is compatible with a given program S and $y(\mathrm{x}) \leq y(\mathrm{x})$ then the output of x after the execution of given program S is not affected by the value of y.

Listing 2.5 shows the language grammar of Volpano, Smith and Irvine which has been proposed for non-interference analysis of sequential program.

Listing 2.5: Grammar of the language of Volpano Smith and Irvine

```
s::= var := exp
        | s; s
        | If exp then s else s end
        | While exp do s
        | skip
```

Where *var* stands for variables, *exp* stands for expressions and S stands for program statements. They provided a syntax directed security type system for annotating all program components including variables, procedure parameters and commands with specific security levels. Figure 2.5 illustrates a typing system equivalent to a security type system of Volpano, Smith and Irvine (2003).

The typing rules $\vdash$ *exp*: τ in Figure 2.5 means that the expression (*exp*) has a type τ. The judgement [*pc*]$\vdash$ S means the program S is typable in *pc* security context. The security level can be *low* or *high*. Considering the rules *E1* and *E2* any expression (*exp*) can have *high* security level and also a *low* security level if *exp* has no occurrence of *h*. In C1, C2 and C3 the rules [*pc*] $\vdash$ skip and [*pc*] $\vdash$ *h*:= *exp* means that Skip and *h*:= *exp* are typable in any security context and *l*:= *low* is typable only if the *expression* has *low* security type. In C4 the judgement [pc]$\vdash$ S1 and [pc]$\vdash$ S2 means that the programs S1 and S2 are typable in the security context

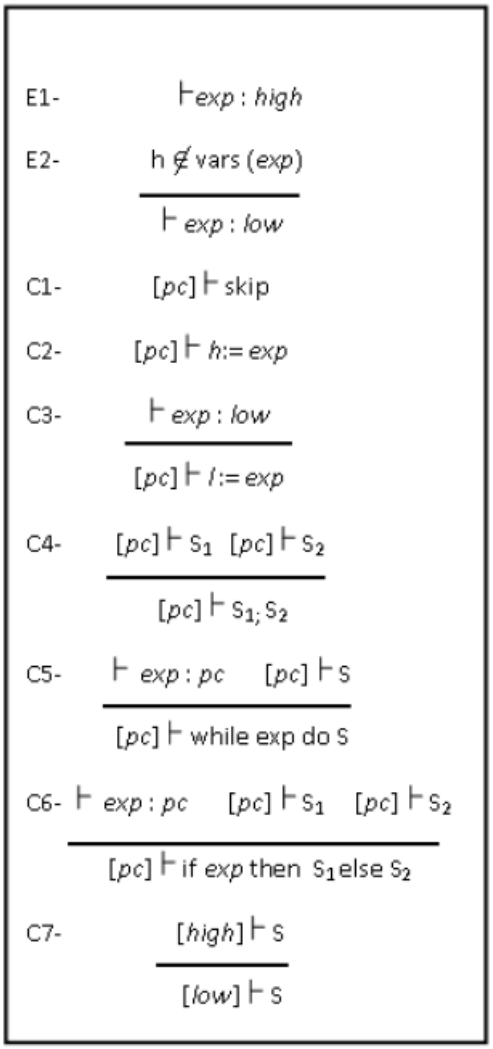


Figure 2.5: Security type system equivalent to the one of Volpano, Smith and Irvine

pc. Rules C5 and C6 considers the *if and loop* statements where all branches should be typable in high security context. The last rule C7 ensures that if the program S is typable in a high security context it leads to the program is typable in a low security context. Another function of rule C7 is that it allows to reset the *PC* to *low* security level after any *high* statement, e.g. loop or condition (Sabelfeld & Myers 2003). Banerjee and Naumann (2002) considered the problem of a sequential object oriented language with Volpano's security type system. Therefore, they extended the Volpano approach to support more sequential object oriented language, e.g.

pointers, private field, inheritance, recursive classes, methods, dynamic binding, class based visibility and type tests where programs have specific security classes. However, these type system approaches are unable to compute the security level associated with each variable in the program neither output nor input. In order to automatically analyse the information flow based on type system it is necessary to include any security mechanism that is able to compute the security level of these variables. Such as the security control mechanism described by Weissman that is able to compute the security level of new created files dynamically (Weissman 1969). In addition, these type system approaches for information flow control are simple to implement, but they are often too imprecise. Consider this sub program:

Listing 2.6: Example of Non-interference

```
low := high;
low := 0;
```

Most type system approaches reject this program based on the directly passing of a *high* security level to a *low* security one as illustrated Figure 2.6, but clearly the program satisfies Non-interference while the output of the *low* level variable does not depend on the value of the *high* level variable. Therefore, the majority of security type system approaches would reject any program with insecure sub-programs.

2.4.2 Semantic Approaches.

The semantic approach is concerned with controlling information flow based on semantic security models that controls information flow in terms of program behaviour (Agat 2000, Sabelfeld & Sands 2000, Pottier & Conchon 2000, Sabelfeld & Sands 2001). Leino and Joshi (2000, 1998) provide a new technique that statically analyse the secure information flow based on a semantic notion of program equality. In their approach they define the equality between two program terms as follows.

S is secure iff (HH ; S ; HH $\doteq$ S ; HH)

They denote program equality by symbol $\doteq$. They wrote (S is secure) to denote that the program S has only secure flow. A key component in their definition is that the high security variable has been assigned to h an arbitrary value in the program which they denote by HH (havoc on h). S is secure if and only if the initial value of any variable with high security level has not any effect on the final value of any variable with low security level in the program. Assume that they denote h to be the variables with high security level and L denote to be low security level variables. Thus, the definition may be described as follows. The occurrence of HH on both sides means that the final value of L is of an interest and an observation of the prefix HH means that both programs are equal based on the output of S (final value of L) independent on the initial value of the high level variables h. Finally, they tried using their definition to prove that the observations of the values (initial or final) of low level variables do not leak information about the initial value of high security level variables. Sabelfeld and Sands (2001, 1999) extended the semantic approach of Leino and Joshi (2000, 1998) to formalize a security specification of secure information flow in sequential program by partial equivalence relations (PERa). The semantics of the program in information flow control can be defined as a mapping over the probability distributions of the information flow in the program. Most of the information flow semantic approaches are semi-semantics because they are not analysis all the possible program behaviours. The advantage of semantic models of secure information flow is that it can be applied to any program structure whose semantics is defined.

2.5 Dynamic Information Flow Analysis

Dynamic information flow analysis attempts to analyze the flow of the information while a program is executing. Dynamic information flow analysis does not require

that all possible paths in the program must be operated or executed. Dynamic information flow analysis always supports a modifiable information flow policy to overcome changeable security requirements which can not be captured statically. Using dynamic information flow analysis it is more easy to handle language features, e.g. arrays, pointers and exceptions than static analysis. Finally and the most important, it is user centric which allows user interaction because the security requirement depends on the type of the user. Despite a long history and a huge amount of research on controlling information flow (Banerjee & Naumann 2005, Volpano et al. 1996, Smith & Volpano 1998, Pottier & Simonet 2003, Fenton 1974*b*), it seems to be very little work done on dynamic information flow analysis and enforcing information flow based policies. Dynamic information flow analysis (Fenton 1974*b*, Brown & Knight 2001, Lam & Chiueh 2006, Birznieks 1998, Vachharajan et al. 2004) has less development than static analysis. Dynamic analysis started very early by the BLP model which attempted to deal with military information flow confidentiality (LaPadula & Bell 1973, Binder et al. 1973). The model aimed to annotate each data element with a security level (*label*) to dynamically control information flow with their two security properties of information flow.

- The simple security property *no read up*
- The star property *no writes down*

Dynamic information flow analysis can be described at machine code level (*binary*), source code level (*program*) and system level (*event*).

2.5.1 Dynamic Analysis at Binary Code Level

Applying dynamic analysis at binary code level does not require the source code of the program. Several work has been done in dynamically ensuring security of data flow. Fenton (1974*b*) proposed the data mark machine which is an abstract model

of implementing memoryless subsystems confinement. In this machine each variable has an associated security class. In his small machine the program counter's PC data mark is computed dynamically for the other storage location is fixed which can be **null** or **priv**. **Null** denotes to non-private (input or output) and **priv** denotes private (input or output) which means that the storage location can only contain (public or secret) information. The data mark machine also includes a stack return address that can be used for program counter declassification. Any return address may be pushed into the stack and any other classified process may then pop the unclassified address into program counter to declassify itself.

Brown and Knight (2001) describes the problem of ensuring secure information flow supported by hardware extension as a set of hardware mechanisms. They practically implemented their model in Hash execution unit (HEX). Simply they add a few hardware mechanisms and software routines to the *Trusted Computing Base* (TCB) to dynamically guarantees that the flow of the information is secure. The main ideas of Fenton (1974*b*) and Brown (2001) is that they enforce a security policy with respect to the lattice structure where $\perp$ is the greatest lower bound or least restrictive class and $\top$ is greatest upper bound or the most restrictive class. Therefore, storing any data value X to a fixed data mark location L requires that the machine checks the data mark security level greater than or equal to $\top$ most restrictive class of the data mark value of X and of the program counter if not the storage operation will be ignored. Vachharajani et al. (2004) addresses an information flow security using **RIFLE**. They designed this architecture to support users interaction to enforce their information flow policy using both hardware extension and binary instrumentation. The main goal of their architectural framework is the users ability to set data policies rather than relying on some one else. The main disadvantage of dynamic information flow analysis at binary code level is the difficulty to deal with implicit information flow. Suppose that there are information

flows created by any piece of software that are not operated or executed. Therefore, many researchers have restricted their work of tracing and monitoring information flow only on explicit information flow.

2.5.2 Dynamic Analysis at Source Code Level

Dynamic analysis of information flow at source code level is easier than the one at binary code level because of the fact that understanding the program binary code to control the flow of the information is harder than understanding the source code. However, there has been a little work done on dynamic information flow analysis at source code level. For example, Birznieks (1998) provided Special mode called (*Perl Taint Mode*) in Perl script language that deals with the notion of taint. In this model each data element is tagged with tainted or untainted security level to prevent users from relying on data that are outside his/her script in order to prevent the operation of any bad commands. Perl taint mode is widely deployed because of the believe that it prevents buffer overflows. However, it is not sufficient to protect confidentiality. Perl also uses taint check (Schwartz et al. 2005) to check perl scripts from any security bugs. When the taint check started all user input will be tagged as tainted. The interpreter will detect any operation that uses taint data which will be considered as an unsafe state of execution that will lead to termination of the execution with an error. Recently, Lam and Chiueh (2006) provided a framework for dynamic taint tracking called *GIFT*. In their framework each variable in the program is associated with a 4-byte tag that can be used to present different type of information such as security class, user ID, file name. These tags are not interpreted by the GIFT compiler but are interpreted by the application programmer. Therefore, the main idea behind the GIFT compiler is to insert a piece of code to call programmer provided tag.

Unfortunately all previous work in dynamic information flow analysis at source code

level did not take in their consideration implicit information flow. Shroff, Smith and Thober in (2007) provided a new approach for dynamic monitor of information flow dependency. The main goal of their approach is to handle any information flow created by a piece of program that is not executed. Their approach is based on the simple idea that implicit information flow appears only if there exist another explicit information flow path. The analysis will get information about all explicit and implicit information flow after an undetermined number of program executions. Therefore, their approach is sound regarding to detecting all information flow types. Of course this approach is of dynamic analysis type but not sufficient to enforce confidentiality from the beginning. Masri and Podgurski (2005) described a new approach of dynamic information flow analysis to detect attacks against a program. In their monitoring approach they tried to detect and prevent any violation of specified information flow policies in multi-threaded program. Thus, their monitoring mechanism does not support configurable information flow policies.

2.5.3 Dynamic Analysis at Event Level

Dynamic information flow analysis at event or system level does not require the study of program code either at source or binary code level. The dynamic analysis at this level attempts to detect and prevent information to flow to any unauthorized part or lower security level in a large system. One of the foundational work on dynamic information flow analysis at event level is Bell and LaPadula Model (1973). This model describes how subjects can write to or read from shared objects. Weissman describes a security control mechanism which dynamically computes the security level of newly created files (Weissman 1969). Weissman provides a security control mechanism that is able to classify the security level of created files dynamically. His mechanism is based on a set of access rights as a theoretical model for security control where each process executing on the machine is associated with a security

level. Nagatou and Watanabe provided a monitoring approach for runtime detection of unauthorized information flow using covert channel in a system that is serving multiple users (Nagatou & Watanabe 2006). In their approach they tried to detect and prevent principals to transfer any sensitive information to unauthorized parts or lower security levels. A covert channel is a mechanism that used to transmit information from one principal of a system to another one. More recently, Guernic in 2009 proposes a dynamic non-interference analysis to enforce confidentiality of secret information at run time (Guernic 2009). His approach is partly dynamic because the approach uses dynamic analysis to detect explicit information flow and static analysis to detect implicit information flow. The approach is restricted to only sequential programs. Thus, the approach did not consider concurrent programs. Cavadini and Cheda presented two information flow monitoring techniques that use dynamic dependence graphs to track information flow during runtime (Cavadini & Cheda 2008). But, their two approaches did not consider the users ability to modify the flow policy at runtime and the security requirements to be dependent on the requirement of individual users and their interaction with the monitoring system.

2.6 Java Bytecode Instrumentation

Several research approaches deal with bytecode instrumentation. For the rest of this section, some existing projects and approaches in the area of bytecode instrumentation will be presented.

Many tools use techniques based on program instrumentation to carry out different tasks such as program tracing and optimization techniques (Srivastava & Wall 1994*a*, 1993). Many tools have been developed using program instrumentation techniques that are used for studying program behavior such as Pixie (Smith 1991), Epoxie (Wall 1992) and QPT (Larus & Ball 1994) that rewrite program executables

to generate address traces and instruction counts. MPTRACE (Susan L. Graham & McKusick. 1993) and ATUM (Agarwal et al. 1986) used techniques to report information about the timing and structure of the program. Also, there are software testing and quality assurance tools that detect memory leaks and access errors such as Purify (Hastings & Joyce 1992) they catch programming errors by using these techniques. Purify inserts instructions directly into the object code produced by existing compilers. These instructions check every memory read and write performed by the program, to report any error. One of the limitations of these tools that they are designed to perform a specific task and are difficult to modify to meet the user needs. Modification of a customized tool to obtain less of more trace information requires a user to access the tool's source code and understand the low level details of the tool. The above mentioned tools operate on object codes for a variety of operating systems and architectures, but none of them works on Java virtual machine class files. There is a tool called NetProf (Parthasarathy et al. 1996) that visualizes Java profile information by translating Java byte codes to Java source code. Srivastava and Eustace provided an analysis tools with OM called ATOM (Srivastava & Wall 1994*b*). ATOM is a framework for building a number of customized program analysis tools. BIT (Lee & Zorn 1997) is a set of interfaces that brings the functionality of ATOM (Srivastava & Eustace 1994) to the Java world by allowing a user to instrument the Java virtual machine class file to observe the runtime behavior of programs for optimization.

Binary Component Adaption provided by (Keller & Horlzle 1998) is a tool that rewrites Java bytecode at load time. The idea of BCA is to add a method to a class to perform symbolic manipulation without doing any operations at the bytecode level. The Java Object Instrumentation Environment *JOIE* (Cohen et al. 1998) is a tool that enables the rewriting or modification of Java bytecode at load time. In fact, most instrumentation frameworks (Parthasarathy et al. 1996, Tatsubori et al.

2001, Tilevich & Smaragdakis 2002) void instrumenting system classes, modifying only user created classes or limit their functionality to only user classes. FERRARI (Binder et al. 2007*b*,*a*) instruments all system classes statically including those that are never used by an application. Chander in 2001 provided a Java bytecode instrumentation mechanism for mobile code security (Chander et al. 2001). This approach replaces each method call that attempts to access any private or secret data with another method call to restrict and limit the functionality. The approach discusses the replacement of target objects with other classes called safe subclasses to implement the security of the mobile code. The obvious limitation of most of the presented techniques is that the bytecode instrumentation process does not cover the execution code of the native methods. (Binder et al. 2006) provides an instrumentation mechanism using a new JVMTI feature in JDK 1.6 called native method prefixing that instruments the native method invocations.

2.7 Information Flow Control Approaches

The field of Information flow control has a long history and a huge amount of research (Banerjee & Naumann 2005, Volpano et al. 1996, Smith & Volpano 1998, Pottier & Simonet 2003, Fenton 1974*b*). Denning in (1975) proposed a secure information flow approach in computer systems. His proposed approach is at compiler time to solve all problems of an implicit information flow. Fentons Data Mark Machine (1974*a*, 1974*b*) was one of the earliest systems that was used for information flow control during runtime to enforce security policies. However, his machine is an abstract machine. **RIFLE'S** architecture (Vachharajan et al. 2004) is provided to be implemented to control information flow security during runtime. Their architecture was designed with the aim of supporting end user choice of the policy decision. Their approach uses a combination of hardware architecture and program binary

translation to trace the flow of the information.

Beres and Dalton (2003) proposed a dynamic instruction stream modification framework to track individual data movements within an application. However, the proposed approach ignored implicit information flow. Chandra in 2006 proposed a hybrid approach that instruments the bytecode with taint propagation code (Chandra 2006) and Haldar (2005) use instrumentation mechanism to enforce security policies. However, both approaches do not consider the native functions call and implicit information flow. Newsome and Song (2005) use the concept of tainting to track untrusted data from potentially unsafe input channels, like networks. However, when detecting attacks a flag will be raised and the execution will be halted. There is no chance of recovery or change of program behaviour.

Other approaches such as Brown and Knight (2001) describe the problem of ensuring secure information flow supported by hardware extension as a set of hardware mechanisms. Their approaches work well for dynamically guaranteeing that the information flow is secure but only if the security class is explicitly specified to high or low.

Another approach of information flow control that should be mentioned is provided by Haffman and Davis (1990). They proposed adding a processor named security pipeline interface (SPI) to be as intermediate between a data host and destination. All data that is send from the host to the destination goes through the SPI. The SPI can analyse, change or delete the data being output. However, using their security pipeline interface they tried to address the integrity of the information flow rather than the confidentiality of the information flow.

2.8 Existing Implementation

In spite of a long history and a huge amount of work, there is a very little practical software that enforces and analyses information flow based policies. This might be due to the restrictions and limitations of many existing mechanism of information flow analysis. But, there are some interesting examples that should be mentioned.

- **JFlow** Myers provided Jif or JFlow which is an extension to the Java language that adds statically checked information flow primitives. It is an imperative language. Jflow compiler works as a source to source translator for Java to check information flow safety. Therefore, its output code can be compiled by any Java compiler. JFlow supports some language features including objects, access control, sub classing, exceptions and dynamic type tests. JFlow uses a decentralized label model where every data item has an associated security label with security policies (Myers 1999).

- **Flow Caml** is an extension of the functional language called Objective Caml language Flow Caml. Flow Caml supports multilevel security with respect to suitable lattice structure where each data item is annotated with aconstant security level for tracing information flow. Flow Caml uses a source to source translator that takes a source code to statically check the flow of the information with respect to the information flow policy as specified by the programmers and then produces Caml code. Its objective Caml code can be compiled by any compiler (Simonet 2003).

- **Bytecode verifier** The Java runtime environment contains a bytecode verifier that enables Java to check untrusted code before running it. The compilation of the source program is to ensure that control flow, type safety and memory

are correct. That may be help of security guarantees (Lindholm & Yellin 1997).

2.9 Program Slicing

Program slicing is an analysis mechanism that was originally provided by Weiser (1979, 1981). The program slicing techniques used while debugging or re-engineering the program. It involves focusing on some parts of a given program that are currently of interest. Weiser used a control flow graph to implement his slicing algorithm as an intermediate representation. There are the following types of program slicing:

- **Forward slicing:** for any statement in the program, all program statements that id may be affected by a given statement (Horwitz et al. 1990). The forward slice concerned with what statements in the program are affected by the variables value in a given statement.

- **Backward slicing:** In this the slice is computed from the point of interest finding all statements in the program that can affect the specified variables at the point of interest and discarding the other program statements. (Herrmann 2001).

- **Static slicing:** is computed symbolically to solve the static analysis problem without considering the input of the program (Weiser 1979).

- **Dynamic slicing:** a slice is calculated for a fixed input or data value. Dynamic slicing is smaller than static slicing, but with restriction of applicability to particular input (Korel & Laski 1988).

Dynamic slicing is closely related to dynamic information flow analysis because both of them are concerned with execution tracing to data dependency and control

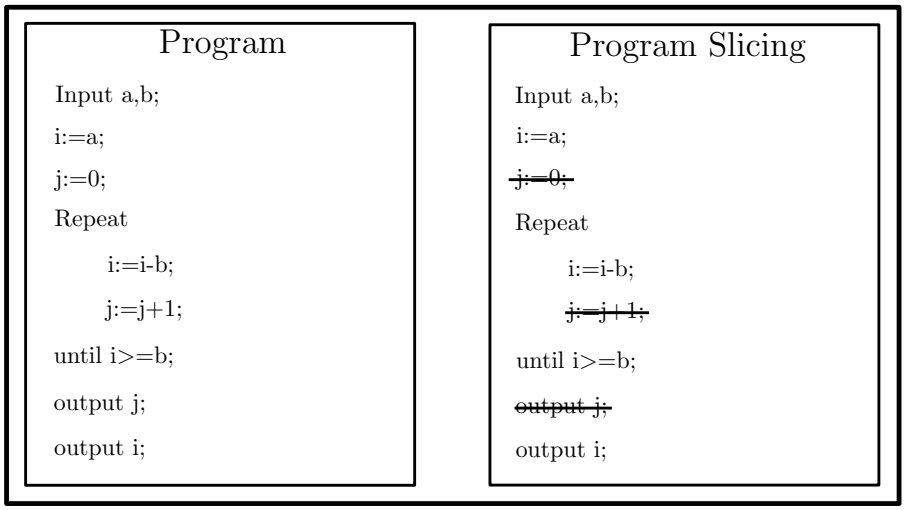


Figure 2.6: Backward slicing of a statement *(output i)*

flow relationship; while, dynamic slicing is focusing on identifying any subset of program statements that influence a particular statement, dynamic information flow analysis is focusing on identifying any subset of program objects that influence a particular object action (Masriand & Podgurski 2005).

2.10 Program Dependencies

The program dependencies techniques are concerned with dependence relations in the program. Dependencies in the program are usually represented by a direct graph between the program statements as nodes for expressions, statements and edges for the dependence relation. To differ between control dependence and data dependence, suppose that the value of expression B controlled by the execution of the statement A then statement A is control dependent on B. If a statement A uses any variable from statement B then statement A is said to be data dependent on B (Ferrante et al. 1987, Cavadini & Cheda 2008). Consider this example below for a program dependence graph.

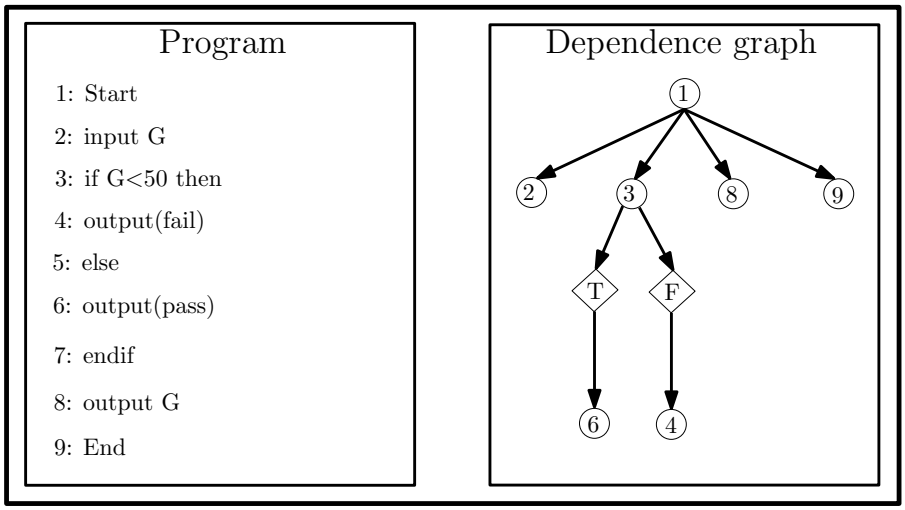


Figure 2.7: Program and its dependence graph

Note the numbered cycles in the graph stands for program statements and square nodes stands for control dependence conditions. From Figure 2.7 one can conclude that.

- Statements 2,3,8 and 9 are guaranteed to execute with respect to control dependencies on statement 1.
- If statement number 3 determines that statement 4 is being executed then statement 4 is control dependent on statement 3.
- In addition, if the same statement 3 determines that statement 6 is being executed then statement 6 is control dependent on statement 3.

Statement 8 is data dependent on statement 2. Program dependencies are closely related to information flow, that is due to the fact that if statement A depends on statement B either direct or indirect then the information flow can occur from B to A. If there is not any type of dependencies between A and B then it can be guaranteed that, there is not any possibility of information flow between A and B.

2.11 Summary

The presented chapter has discussed the related research of this thesis. It has provided an introduction to the principles of security, static information flow analysis and dynamic information flow analysis. It has provided an overview of existing approaches of bytecode instrumentation and information flow control and discussed the difficulties and problems of the related research. The following are some common problems of the previous mentioned works in this chapter either static or dynamic information flow analysis.

- Fixed information flow policy.
- Works well for sequential programs, but not for concurrent ones. Concurrence refers to programs that are executed in parallel.
- The previous information flow security mechanisms are not usable in practice.
- The user is unable to modify the information flow policy during runtime in response to incidents.
- The behaviour of a program leaking confidential information can not be alter by the monitor according to the user decision.

Chapter 3

Architecture

Objectives

- General overview of the proposed framework.
- Describe the framework architecture.
- Show how the framework components interact.

3.1 Introduction

This chapter provides a general overview of the proposed framework and describes the framework architecture. In this chapter the focus is on how each component of the proposed framework interacts to trace and control the information flow within a Java application. The remainder of this chapter is structured as follows. Section 3.2 provides general overview of the framework. Section 3.3 provides a brief description of the security requirements specification. Section 3.4 describes the information flow policy. Section 3.5 introduces the assertion points. Section 3.6 provides a brief description of the event recognizer. Section 3.7 provides a brief introduction to the runtime checker component. Finally Section 3.8 describes the user feed back component.

3.2 General Overview of the Framework

In static program analysis all possible paths of the program execution must be free of any illegal information flow. If any illegal information flow is detected then the static analysis mechanism will reject the whole program as insecure.

Graphically we can depict the set of possible program behaviours by a blank circle and the set of all insecure program behaviour (defined in the policy) by a hashed circle. In these terms a program is rejected by static analysis if the intersection of both is not empty. In Figure 3.1 we depict the case for dynamic information flow analysis. Consider that a program is in a state 0 and performs an operation α that causes an information flow. We can distinguish two cases:

1. After the execution of α the program is in a secure state.

2. After the execution of α the program is in an insecure state.

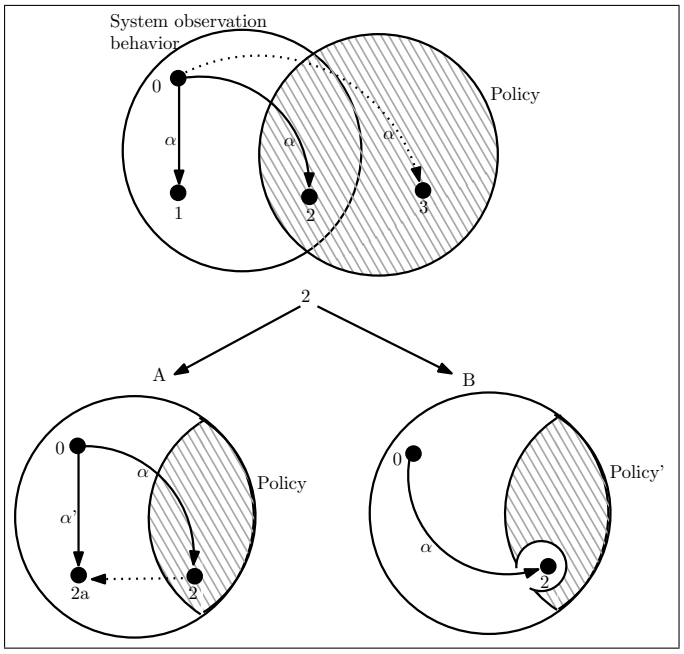


Figure 3.1: Monitoring mechanism

The hypothetical third case, that the program exhibits a behaviour that is defined by the policy as insecure, but is outside of the set of possible behaviours, can be ignored. In our framework we check whether the program is about to enter an insecure state by intercepting the operation α.

- In case 1, that α leads to another secure state the program will be allowed to perform α.

- In case 2, the runtime monitoring mechanism will send feedback to the user asking about the violation of information flow.

The user has two options on how to proceed:

A. The user changes the operation α to another operation α' so that the resulting state is secure with respect to the policy. Such changes can for example be the termination of the program or (manually) sanitizing the information that flows in α.

B. The other option is to modify the Policy into a Policy' such that α leads to a secure state. This could for example be introducing a one-off exception for the current flow or defining a separate context in which the information flow is considered legal.

Our approach is based on the observation of information flow during application run time. The user feedback component handles all interactions with the system and the user. It runs in a separate thread of control so that user interaction can be overlapped with information flow control. The user feedback component also allows the user to administrate the policy.

When the software is running, the user feedback component receives feedback from the runtime checker (Steering). If the software is about to enter an insecure state then the user will be asked to determine whether the information flow should be aborted or allowed to flow and continue under a modified policy as illustrated in Figure 3.2. For example given a policy that states that Bob's password must not be

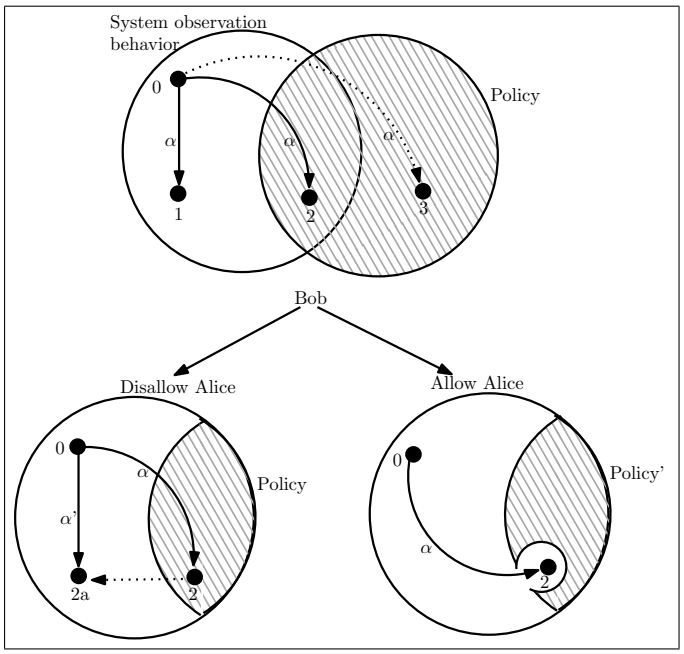


Figure 3.2: Monitoring mechanism

shared with any other user (Alice and Eve). If Bob now wants to give his password to Alice to perform some function on his behalf, our approach will detect this violation of information flow and ask Bob how to proceed. Bob can then choose to stop the operation that would violate the flow (i.e. Alice does not obtain the password) or he allows this flow and changes the policy accordingly to reflect this decision. Moreover, this change can be temporary (a one of exception) or permanent (He can pass the password again to Alice).

Security will be achieved by an interactive process in which our framework will query the user for security requirements for specific information that are made available to the software and then continue to enforce these requirements on the application using runtime verification technique for tracing information flow. The assertion points will be inserted in the application bytecode and the event recognizer receives low level information from the assertion points and sends any event that attempts to make information flow to the runtime checker. The runtime checker checks the received event against the flow policy and sends a feedback to the user throughout the user feedback component.

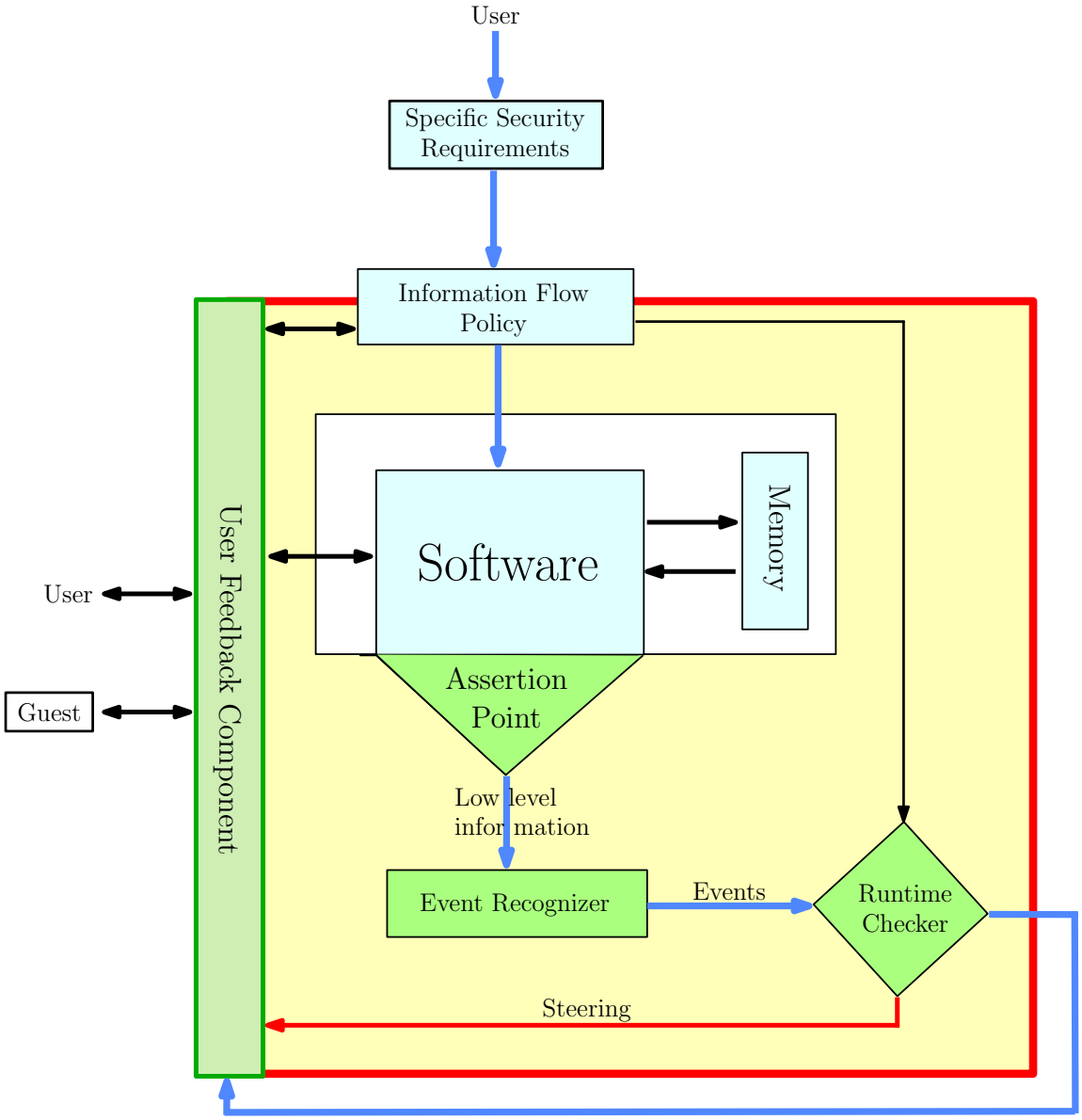


Figure 3.3: Runtime verification of information flow

As illustrated in Figure 3.2 the runtime verification of information flow framework consist of several components:

3.3 Security Requirements Specification

Stakeholders normally have a number of concerns that come with their desired system and are typically high-level strategic goals. In this component the stakeholders specify the desired characteristics that a system or subsystem must possess with respect to sensitive information flow. In our previous example Figure 3.2, this is the requirement that Bob's password must not be shared with any other user (Alice and Eve). The stakeholders should provide the *source* of the information and the output channel a *destination* which will be formally expressed in the information flow policy, to enable our runtime monitoring mechanism to (dis)allow the information flows from source to destination. Such properties can ensure the confidentiality of the information flow.

3.4 Information Flow Policy

An information flow policy is a type of security policy that defines the authorized paths or the way information moves throughout the system (Bishop 2003) as obtained from the stakeholders, which can be a set of rules, axioms and laws that are provided to regulate how information must flow to prevent leak of sensitive information. The information flow policy is designed to preserve information confidentiality. In our framework, the information flow policy expresses the security requirements as specified by the stakeholder/user as a set of rules that are understandable by our runtime monitoring mechanism. The goals of the information flow policy is to prevent secret information from flowing to a user or client software not authorized

to receive it. For example, an information flow policy can state more context dependent flows, such as (Bob's password can only flow to Alice if Bob explicitly agrees). Chapter 6 describes the notion of information flow policy in detail.

3.5 Assertion Points

Assertion points are program fragments as a collection of probes that will be added into the target software. The main functionality of the assertion point is to send the relevant state information to the event recognizer. This will ensure the monitoring of relevant objects during the execution of the software. The probes are inserted into all locations where monitored objects are updated such as (program variables and function calls); unlike traditional runtime verification approaches our assertion points are inserted before the operation to be able to intercept updates and thus prevent the system from entering an insecure state. In order to send state information to the event recognizer our framework uses a novel instrumentation mechanism (Chapter 4) to insert assertion points.

3.6 Event Recognizer

The event recogniser is the part of the framework that detects an event from the information received from the assertion point. The event recognizer is used as a communication interface between the assertion points and the runtime checker. The event recognizer sends any event that attempts to change the state of the software to the runtime checker (according to the information flow policy). Although, it is possible to combine these two components the assertion point and the event recognizer, we separated them to make the implementation of our architecture more extensible and to minimize the interference with the monitored application, such as allowing

the integration of several runtime checkers that verify different type of requirements. For example, the management of obligations related to information flow could be placed in a logically separate component. Chapter 5 provides a description of how the events will be recognized and sent to the runtime checker.

3.7 Runtime Checker

The runtime checker component checks whether the execution trace belongs to the set of all acceptable behaviours as defined by the security requirements specification and expressed in the information flow policy. The runtime checker verifies and decides whether or not the current execution trace as obtained from the event recognizer satisfies the information flow policy and sends feedback to the user feedback component when it determines that the software is about to enter an insecure state, e.g. any information flow that violates the policy. Chapter 5 describes how the runtime checker checks the information flow policy and sends a feedback to the user feed back component.

3.8 User Feedback Component

The user feedback component is an interface between the system and the user. An essential functionality of the user feedback component is that all user interaction passes through this component. The user feedback component informs the user about policy violations detected by the runtime checker. As illustrated in Figure 3.1 if the runtime checker determined that this state execution would violate the information flow policy then it sends feedback to the user, the system behaviour will be changed accordingly, or the policy will be modified according to the user decision. The user feedback component is described in Chapter 6.

3.9 Summary

The presented chapter has provided a brief overview of the proposed framework architecture. It describes all components of our framework and identifies the technology that will be used in the framework. The following chapters give the specification of each component of the presented framework. An instrumentation mechanism for inserting different type of assertion points is provided in Chapter 4. The event recognizer and runtime checker algorithms are given in Chapter 5. The definition of the information flow policy and user feedback component is provided in Chapter 6.

Chapter 4

Bytecode Instrumentation

Objectives

- Introduction to Java bytecode and virtual machine.
- Class loading, linking and initialization.
- Instrumentation algorithm.
- Instrumentation examples.

4.1 Introduction

This chapter provides an introduction to Java bytecode, class loader in Java virtual machine and how a Java class file will be instrumented in order to monitor and control the information flow within a Java application. All computations in the Java virtual machine are centred on the stack. Due to that the Java virtual machine do not deal with registers for storing any arbitrary values, whereas everything is pushed onto the stack frame of the current executed method before it can be used in the computation. The focus in this chapter is on the loading and instrumentation of the target program. This chapter also provides examples of Java bytecode instructions and how they are instrumented. Our approach consist of two main steps:

- Loading and Instrumentation of class files of the target program.
- Execution of the target program and monitoring the information flow with respect to the information flow policy.

Figure 5.1 shows a flowchart of our runtime monitoring mechanism.

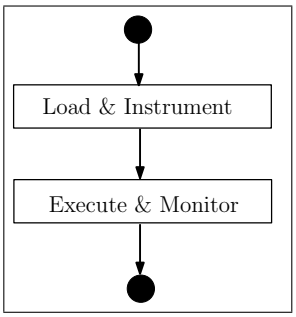


Figure 4.1: Monitoring mechanism flow chart

The bytecode instrumentation is a process that inserts method calls in the bytecode, such that information can be extracted from the target program while it is being executed. The remainder of this chapter is structured as follows. Section 4.2 introduces Java bytecode. Section 4.3 provides an example of a Java source file and

its bytecode format. Section 4.4 describes the class loader in Java virtual machine. Section 4.5 explores the loading phase. Section 4.6 describes the linking process phase. Section 4.7 describes the initialization process. Section 4.8 describes the instrumentation of assertion points and provides examples of Java bytecode instructions before and after instrumentation. Section 4.9 shows the instrumentation of explicit information flow instructions. Finally Section 4.10 shows the instrumentation of implicit information flow instructions.

4.2 Bytecode

Bytecode is object code that is the result of program compilation, usually executed by a virtual machine, rather than by hardware. The virtual machine converts each bytecode instruction into specific machine instructions that are understandable by the computer's processor. Hence, Java bytecode makes it possible for Java code to run on many different platforms. Each instruction of bytecode has a size of one byte (Lindholm & Yellin 1997, Meyer & Dowing 1997). This bytecode representation of a Java source file is also known as a Java class file. The bytecode (Opcode) of each Java instruction is provided in Appendix A.

4.3 Example: Java Bytecode

This section provides an example of a Java source file and its corresponding class file after compilation. Listing 4.1 represents a Java source file before converted into Java bytecode (class file).

Listing 4.1: Source file of Add.java

```
1 public class Add {
2     public static void main(String[] args) {
3         int i=2;
```

```
4         int j=3;
5          System.out.println(i+j);
6      }
7  }
```

Listing 4.2 shows the class file (Generated bytecode) of the Java source file presented in Listing 4.1.

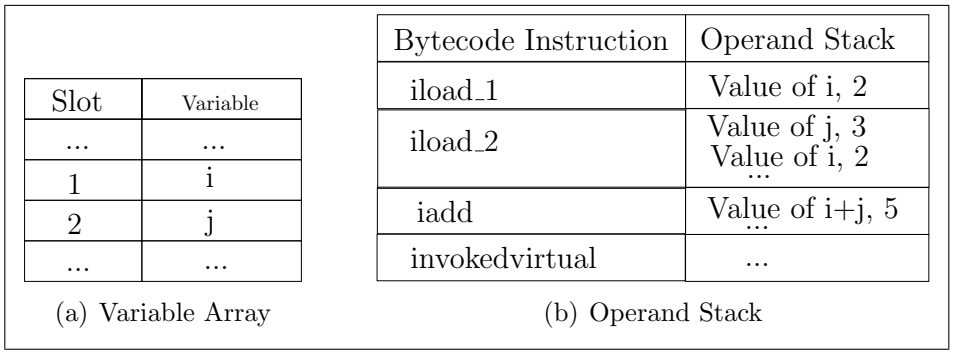
Listing 4.2: Class File of Add.java

```
0: iconst_2                    // push int constant 2 onto the stack
1: istore_1                    // pop 2 into local variable position 1,
2: iconst_3                    // Push int constant 3 onto the stack
3: istore_2                    // pop 3 into local variable position 2,
4: getstatic #2 =Field java.lang.System.out(Ljava/io/PrintStream;)//push field onto the stack.
7: iload_1                     // push 2 from local variable position 1 onto the stack
8: iload_2                     // push 3 from local variable position 3 onto the stack
9: iadd                        // pop top two values from stack, add and push the result onto stack
10: invokevirtual #3 = Method java.io.PrintStream.println((I)V)// Call println method, print i+j
13: return                     // Return void.
```

In Java bytecode each invoked method has a corresponding bytecode array. These bytecode values correspond to the index within the bytecode array where each opcode and its arguments are stored. The numbers that appears on the left of each bytecode instructions line (0,1,2,3,4,7,8,9,10 and 13) are the index of each instruction in the opcode array that contains the bytecode of the Java virtual machine for this method.

As illustrated in Listing 4.2 the opcodes indexes are not sequential. The opcodes indexes are not sequential because some of the opcodes have parameters that take up space in the bytecode array, e.g. the *iconst_2* instruction has no parameters and it occupies one byte in the bytecode array. Therefore, the next opcode, *istore_1*, is at Opcode index 1. The opcode, *getstatic* is at index 4. However, *iload_1* is Opcode index 7 because the *getstatic* opcode and its parameters occupy location 4, 5, and 6. Index 4 is used for the *getstatic* opcode, Opcode indexes 5 and 6 are used to hold

the *getstatic* opcode parameters. These parameters are used to construct an index into the runtime constant pool for the class reference, where its value is stored. In JVM each method has its own stack frame. Such a frame is composed by a local variable array that contains all the local variables used in the current method and by the operand stack to execute the bytecode instruction. Considering our example Listing 4.1, a snap shot of the local variable array and the operand stack required to execute such program are illustrated in Figure 4.2.



(a) Variable Array (b) Operand Stack

Figure 4.2: JVM - Bytecode instruction execution

An important feature of the JVM is that the bytecode instructions are typed. In the first four Opcode indexes, index 7 and 8 (*iconst_2, istore_1 and iload_1*) as shown in Listing 4.2, the prefix is a representative of the type that the opcode is working with (prefix *i* means integer). Other opcodes prefixes are (*a*) for object reference (*b*) for byte, (*c*) for char, (*d*) for double, etc. This prefix gives immediate knowledge about what data type is being manipulated. The iadd instruction in Opcode index 9 requires that the stack initially contains at least two elements and these two elements are of type int. Then *iadd* pushes back a result of type *int* onto the stack.

Similarly, for the *getstatic #2* instruction at Opcode index 4 in Listing 4.2. The *#2* represents a runtime constant pool index with a hash sign and follow the instruction is a comment identifying the runtime constant pool item referenced to access the instance field *out* of type *Ljava/io/PrintStream;* declared in the class

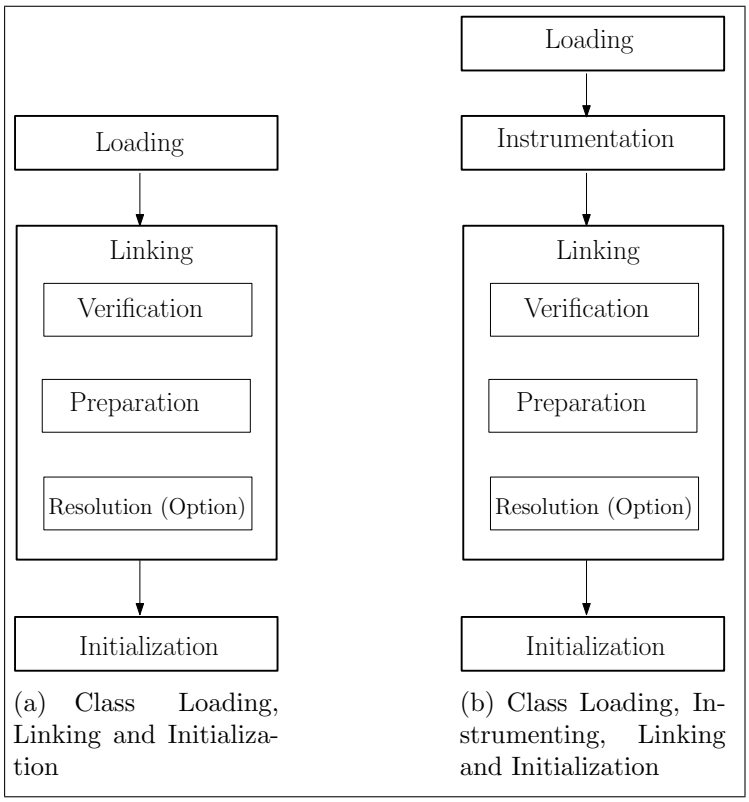
Java.lang.System. This opcode requires that the top of the stack contains a reference to an instance of class *Java.lang.System*. Then the value of the class is pushed onto the stack. The same is true for all other Java virtual machine instructions that are prefixed by a hash sign. More generally, in JVM the code must meet the following conditions:

- Type correctness, instruction arguments must be of expected type.
- No stack overflow or underflow.
- Code containment, the program counter must always point to the appropriate instruction.
- Register initialization, a load from a register must follow at least one store in this register.
- Object initialization, when an instance of a class is created, one of the initialization method of this class must be invoked.
- Access control, class reference, method invocation and field access must respect the different types (public, protected, private, etc) of class, method or field.

4.4 Class Loader in Java Virtual Machine

A Java program is composed of many individual class files, each of which corresponds to a single Java class. These class files are loaded as needed by the program (DeveloperWorks 2001, Liang & Bracha 1998), by a special class called a class loader. The concept of the class loader is responsible for the loading, linking and file system interaction transport to JVM. This means that the class files can be loaded from remote locations, e.g applets loaded their classes via *HTTP*, without requiring a change in the JVM. The loading, linking and initialization of classes or interfaces

used by an application are necessary steps before any application is executed by JVM (Meyer & Dowing 1997, Lindholm & Yellin 1997). Figure 4.3 shows the JVM processes.



(a) Class Loading, Linking and Initialization

(b) Class Loading, Instrumenting, Linking and Initialization

Figure 4.3: Processes in JVM

4.5 Loading

Loading is the first operation performed by the class loader. The aim of the loading process is to obtain a binary form of a class file (Meyer & Dowing 1997, Lindholm & Yellin 1997). The main components of this process are the Java Class file and the Class loader.

4.5.1 Java Class File

A Java class file contains all necessary information that are used by JVM to define a class or interface during runtime:

- Bytecode of the methods declared and implemented in a Java class.
- Symbolic reference to the super class of the class defined by a class file.
- List of all fields defined in the class.
- Constant pool containing literals, descriptor of methods and fields declared in or used by the class.
- Other information of a Java class such as local variable table and exception table.

4.5.2 Class Loader

The main aim of this component is to dynamically load and instrument a class represented by a given class file. To generate a complete trace during execution, the class loader is not only expected to load a class represented by a given class file, but all classes used by a class corresponding to the given class file (Chiba 2007, Liang & Bracha 1998). Generally, whenever a class is used for the first time the class loader will load, link and initialize it.

4.6 Linking

In the linking process, the binary form of a class or interface is converted into a runtime state for use within the JVM. This phase is divided into three different steps (Meyer & Dowing 1997, Lindholm & Yellin 1997).

4.6.1 Verification

Verification is the first step of the linking process. The main aim of the verification step is to ensure that the binary representation of the given class file (Bytecode) is structurally valid. To ensure that every bytecode instruction has a valid opcode, each method has a correct signature and that the bytecode instruction sequence does not violate the type discipline of Java (Meyer & Dowing 1997, Lindholm & Yellin 1997), e.g. if the bytecode tries to load an integer value from the local variable array using a bytecode instruction for loading float values then the verifier will throw an exception as it will violate the type discipline of Java.

4.6.2 Preparation

During preparation, the JVM will create static field for the class or interface and then initialize them with their default values. This step is only the preparation of an execution, thus, no virtual machine code is executed, yet.

4.6.3 Resolution

During resolution, the Java virtual machine will replace all the symbolic references to class and fields stored into the constant pool to actual references. It is not mandatory to resolve a symbolic reference until it is used for the first time, it is up to the implementation to decide when to resolve symbolic references (Lindholm & Yellin 1997, Venners 1999, Liang 1999).

4.7 Initialization

Initialization is the process of executing defined class static initializers and initializers for static fields. In case of an interface only initializers for static fields are executed

(Lindholm & Yellin 1997). The initialization process has two steps:

1. Initialize the direct super class of the class (if not initialized already)

2. Execute the class or interface initializer (if present)

4.8 Assertion Points

Assertion points are program fragments that act as a collection of probes (filters) that will be inserted into the class file. The essential functionality of an assertion point is to send pertinent state information to the event recognizer. This will ensure monitoring of relevant objects during the execution of the software. Our assertion points are inserted before the monitored bytecode instruction to be able to intercept updates and thus prevent the system from entering an insecure state as specified in the information flow policy.

4.8.1 Overview of Instrumentation Process

Instrumentation is an effective technique against well specified attacks, which include denial of service and information leaks via specific pathways from source to destination (Chander et al. 2001). Bytecode instrumentation is a process to insert assertion points in the bytecode, that should be performed after loading and before linking and initialization of the class file as show in Figure 4.3(b).

The following presents the technical details of our instrumentation process. Our instrumentation process illustrated in Figure 4.4 uses a special Java library called *Java agent*.

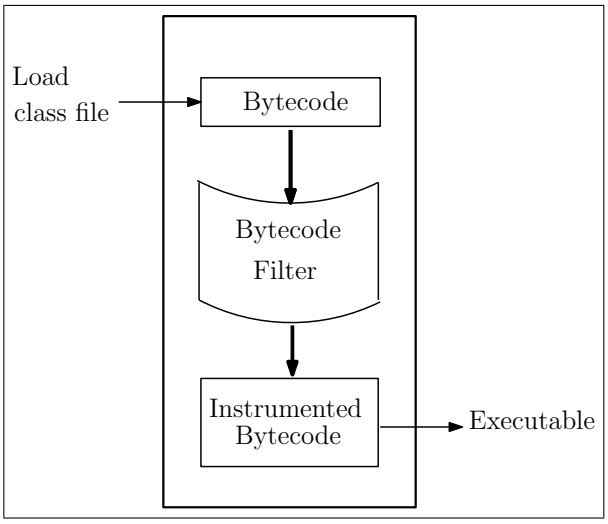


Figure 4.4: Instrumentation process

4.8.1.1 Java Agent

A Java agent is a pluggable library that runs embedded in a Java virtual machine and intercepts the class loading process. This allows our instrumentation process to monitor the class loading process and instrument (insert assertion points) the bytecode of the classes to provide the required information. Our instrumentation process uses the *Java agent* and *Javassist* open source because the core Java packages do not have support for programmatic manipulation of bytecode.

The agent class implements a public static *premain* method similar in principle to the main application entry point. After the Java Virtual Machine has initialized, each *premain* method will be called in the order the agents were specified, then the real application main method will be called. The *premain* method allows the instrumentation process to manipulate classes in two ways: class file transformation and class file redefinition.

- Class file redefinition is used to redefine (replace) the definition of already loaded classes.

- Class file transformation. Registering a transformer allows that all future class

definitions will be seen by the transformer. The transformer is called when classes are loaded and redefined.

The bytecode instructions are divided into 20 different categories, according to their relation with the information flow at bytecode level. Table 4.1 illustrates these instructions categories. The categorization is based on the instruction manipulation of the operand stack. Our instrumentation process considers the following constraints to ensure the class file validation.

1. The stack can not overflow or underflow.
2. The operands will always be the correct type.
3. Branches will be within its bounds of the code array for the method.
4. The target addresses of all control flow instructions are points to the start of an instruction.
5. No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates.
6. All references to the constant pool must be to an entry of the appropriate type.
7. The code does not end in the middle of an instruction.

4.8.2 Bytecode Filters

The bytecode filter analyses the loaded class file and inserts assertion points prior to the verification and linking of the bytecode. In this way, JVM only receives bytecode that has been analysed. The essential functionality of the filter is to keep track of all application objects, such as program variables and function calls to send pertinent state information to the event recognizer. Instrumentation can be

Category	Instruction	Description
Const	aconst_null, iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5,lconst_0, lconst_1, fconst_0, fconst_1, fconst_2, dconst_0, dconst_1, bipush, sipush, ldc, ldc_w, ldc2_w	This category includes all instructions that pushed constant onto operand stack
Load	iload,lload,fload,dload,aload,iload_0,iload_1, iload_2,iload_3,lload_0,lload_1,lload_2,lload_3, fload_0,fload_1,fload_2,fload_3,dload_0,dload_1, dload_2,dload_3,aload_0,aload_1,aload_2,aload_3	Push the local variable value onto the operand stack
Store	istore,lstore,fstore, dstore, astore, istore_0, istore_1, istore_2, istore_3, lstore_0, lstore_1, lstore_2, lstore_3, fstore_0, fstore_1, fstore_2, fstore_3, dstore_0, dstore_1, dstore_2, dstore_3, astore_0, astore_1, astore_2, astore_3	Pop value from the top of operand stack, and store it in the local variable.
Astore	iastore, lastore, fastore, dastore, aastore, bastore, castore, sastore	Pop the arrayref, index, and value from the operand stack.
Pps	pop, pop2, swap	Pop 1 or 2 values on the top of the operand stack.
Dup	dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2	Duplicates the value on the top of the operand stack
Return	ireturn, lreturn, freturn, dreturn, areturn, return	pop the return value from stack
LoadField	getfield, getstatic	Fetch the objectref value and push it onto the stack
Storefield	putfield, putstatic	Pop the top stack value
Union	iadd, ladd, fadd, dadd, isub, lsub, fsub,dsub, imul, lmul, fmul, dmul, idiv, ldiv, fdiv, ddiv, irem, lrem, frem, drem,ishl, lshl, ishr, lshr, iushr, lushr, iand, land, ior, lor, ixor, lxor, iaload, laload, faload, caload, saload, arraylength,daload, aaload, baload,	Pop two values from the stack and push the result onto stack
New	new	Push the objectref onto the stack
NewArray	newarray, anewarray, multianewarray	Push the arrayref onto the stack
Monitor	monitorenter,monitorexit	Pop the objectref from the stack
Invoked	invokevirtual, invokespecial, invokestatic, invokeinterface	Pop the argument and objectref from the stack. In invokestatic the argument only popped
Ifcond	ifeq, ifne, iflt, ifge, ifgt, ifle, ifnull, ifnonnull	Pop the top value on the stack and compare it against zero
Ifcmp	if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt,if_icmple, if_acmpeq, if_acmpne	Pop 2 values on the top of the stack and compare them.
Endif	goto,, goto_w,	Pop the objectref from the stack
Switch	tableswitch, lookupswitch	Pop int value from the stack.
Out of scope	ineg, lneg, fneg, dneg, iinc, i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c, i2s, lcmp, fcmpl, fcmpg, dcmpl, dcmpg, jsr, jsr_w, ret, wide, checkcast, instanceof, breakpoint, impdep1, impdep2,	Instructions in this category have no implication for information flow

Table 4.1: Instruction categories

done either statically or dynamically. Static instrumentation means inserting the assertion points after loading and before linking the class file Figure 4.3(b), whereas dynamic instrumentation involves inserting the assertion points at runtime.

Our runtime monitoring mechanism uses static instrumentation to avoid incur extra overhead at run-time to determine when it is secure to insert and remove filters. Also, dynamic instrumentation requires a complex instrumentation mechanism (Miller et al. 1995). Instrumentation can be done in two ways. The user has access to the source code of the program, then inserts filters into the system according to the source code. The automatic instrumentation determines what assertion points are suitable for each bytecode instruction to be inserted. Our approach uses automatic instrumentation however, the weak points of automatic instrumentation are that it may not be easy to define an event of high level behaviour based on low level state information and it may not be easy to handle complex programs. All classes being loaded within an application will be instrumented in terms of keeping track of all application objects.

4.8.3 Class Instrumentation

Java programs are composed of classes. Classes are composed of members. Members are either fields or methods. The purpose of class instrumentation is to instrument bytecode of a given class file such that the instrumented class file produces required trace information to be sent to the event recognizer.The class instrumentation algorithm is in Listing 4.3.

Listing 4.3: Class instrumentation algorithm

```
Let Mc be the set of methods detected in the class C.
Let Super(C) be the super class of C.
    Load(C):
            Read bytecode of C
            If Super(C) is not loaded then
```

```
        load(Super(C))
    else
      For each m in Mc
        if m is not native then
          Instrument Method(m)
```

Listing 4.3 shows the class instrumentation algorithm that loads class and its super classes. The algorithm also stated that all not native methods in class and its super classes will be instrumented.

4.8.4 Method Body Instrumentation

The method body instrumentation instruments the byte code of the method in such a way that whenever an instruction of the method causes an information flow, then the assertion point sends the required trace information to the event recognizer.

Listing 4.4: Method instrumentation algorithm

```
Step 1: Get method m code.
Step 2: for each opcode x in m
        x is Const:                     Instrument const.
        x is Load:                      Instrument load.
        x is Store:                     Instrument store.
        x is Astore:                    Instrument astore.
        x is Pps:                       Instrument pps.
        x is Dup:                       Instrument dup.
        x is Return:                    Instrument return.
        x is Loadfield:                 Instrument loadfield.
        x is Storefield:                Instrument storefield.
        x is Union:                     Instrument union.
        x is New:                       Instrument new.
        x is Newarray:                  Instrument newarray.
        x is Monitor                    Instrument monitor.
        x is Invoked method
          x is native:
            x is NativeWrite :          Instrument write.
            x is NativeMethod:          Instrument native.
          x is method:                  Instrument method.
```

```
          x is constructor:                Instrument SpecialMethod.
        x is Ifcond:                       Instrument ifcond.
        x is Ifcmp:                        Instrument ifcmp.
        x is Switch:                       Instrument switcth.

Step 3: Get method m code.
Step 4: for each opcode x in m
        x is Ifcond:                       Instrument Endif.
        x is Ifcmp:                        Instrument Endif.
```

The method body is a sequence of opcodes. The method instrumentation algorithm categories the method opcodes and shows how to examine each opcode to determine in which opcode category it belongs as indicated in Table 4.1. In order to send appropriate state information the method instrumentation algorithm has two iterations. The first iteration is performed by steps 1 and 2 of Listing 4.4 to instrument all types of Java bytecode. The second iteration is performed by steps 3 and 4 of Listing 4.4 to instrument *Endif* assertion point. Because Java bytecode instructions do not have end if statement. In addition, the offset address of the *if* conditions may changes while method instrumentation as will be shown in Examples 1 and 2 of Section 4.10.4.

The method instrumentation algorithm has different subcategories for invoked method such as native write method and normal method otherwise instrumented as native method invocation. This distinction allows for capturing the required code information to enable event recognizer and runtime checker to control the information flow. Our instrumentation mechanism has divided the instrumentation of the instructions into two types.

- Explicit information flow instrumentation to control the explicit flow of the information.
- Implicit information flow instrumentation to control the implicit flow of the information.

The algorithm in Listing 4.4 is performed for all class methods. The categories and subcategories in the method instrumentation algorithm of Listing 4.4 are explained in detail in the next sections by showing each instruction before and after instrumentation.

4.9 Explicit Information Flow Instrumentation

In Java bytecode, explicit information flow occurs with an assignment statement and may occur with the method invocation. The next subsections show the instrumentation of the categories that deal with the explicit information flow indicated in the method instrumentation algorithm (Listing 4.4).

4.9.1 Instrument const

Const comprises all instructions that push a constant value onto the top of the stack as illustrated in Table 4.1. All types of this category will be instrumented as follows:

Listing 4.5: Constant instrumentation

Before instrumentation:

```
iconst_1
```

After instrumentation:

```
invokestatic #387 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)
iconst_1
```

Listing 4.5 indicates that *const* assertion point involves one line of opcode that inserted to call const() method in the event recognizer.

4.9.2 Instrument load

All instructions that belong to category load in Table 4.1 push one value from the local variable array to the top of the stack will be instrumented as follows. Listing 4.6 indicates that *Aload* assertion point involves three lines of opcode that inserted to duplicate the loaded object, push label 1 into the stack and calling Aload() method in the event recognizer.

Listing 4.6: Load instrumentation

Before instrumentation:

```
aload_1
```

After instrumentation:

```
aload_1
dup
ldc_w #352 = "1"
invokestatic #354 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Aload((ILjava/lang/String;)V)
```

4.9.3 Instrument store

Store involves all instructions that pop values from the current method stack and store it in the local variable table as indicated in Table 4.1. Store instrumentation is as follows:

Listing 4.7: Store instrumentation

Before instrumentation:

```
astore_2
```

After instrumentation:

```
dup
ldc_w #514 = "2"
invokestatic #517 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store((Ljava/lang/Object;Ljava
    /lang/String;)V)
```

```
astore_2
```

Listing 4.7 indicates that *Store* assertion point involves three lines of opcode that inserted to duplicate the loaded object, push label 2 into the stack and calling Store() method in the event recognizer.

4.9.4 Instrument astore

Astore includes all instructions that pop three values from the top of the current method stack and store them as an array. These popped three values represent (Array object reference, value and an index in the array). Astore instrumentation is as follows:

Listing 4.8: Astore instrumentation

Before instrumentation:

```
iastore
```

After instrumentation:

```
invokestatic #125 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Astore(()V)
iastore
```

Listing 4.8 indicates that *Astore* assertion point involves one line of opcode that inserted to call Astore() method in the event recognizer.

4.9.5 Instrument pps

Pps comprises three instructions that are provided for the direct manipulation of the stack as indicated in Table 4.1.

- Pop: pops the top value from the stack.
- Pop2: pops the top one or two values from the stack.

1. Pop one if the value is of type long or double.
2. Pop two if the value is of type (Boolean, byte, char, short, int, float, reference or returnAddress)

- Swap: swaps the top two values on the top of the stack.

The instrumentation is as follows:

Listing 4.9: Pop instrumentation

Before instrumentation:

```
pop
```

After instrumentation:

```
iconst_1
invokestatic #252 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Pps((I)V)
pop
```

Listing 4.9 indicates that *Pps* assertion point involves one line of opcode that inserted to call Pps() method in the event recognizer.

4.9.6 Instrument dup

Dup instructions are provided for the direct manipulation of the stack values as described in Table 4.1.

The instrumentation is as follows:

Listing 4.10: Dup instrumentation

Before instrumentation:

```
dup
```

After instrumentation:

```
iconst_1
iconst_0
invokestatic #112 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Dup((II)V)
dup
```

Listing 4.10 indicates that *Dup* assertion point involves one line of opcode that inserted to call Dup() method in the event recognizer.

4.9.7 Instrument return

Return comprises five instructions that pop values from the method stack as return values and one instruction *return* that does not pop any value from the stack because its method does not has a return value.

The instrumentation is as follows:

Listing 4.11: Return instrumentation

Before instrumentation:

return

After instrumentation:

iconst_1

invokestatic #117 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Return((I)V)

return

Listing 4.11 indicates that *Return* assertion point involves two lines of opcode that inserted to push constant 1 into the stack and calling Return() method in the event recognizer.

4.9.8 Instrument loadfield

Loadfield includes two instructions *getfield* and *getstatic*. Our instrumentation process prefixes fields name with 0 to distinguish between local variables and fields as shown in Listing 4.12.

The instrumentation is as follows:

Listing 4.12: Load field instrumentation

Before instrumentation:

getstatic #35 = Field java.lang.System.out(Ljava/io/PrintStream;)

After instrumentation:

ldc #205 = "0out"

invokestatic #207=Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.LoadField((Ljava/lang/String;)V)

`getstatic #35 = Field java.lang.System.out(Ljava/io/PrintStream;)`

Listing 4.12 indicates that *LoadField* assertion point involves two lines of opcode that inserted to push label 0out into the stack and calling LoadField() method in the event recognizer.

4.9.9 Instrument storefield

Storefield involves two instructions *putfield* and *putstatic*. Again 0 will be added to the left of the field name.

The instrumentation is as follows:

Listing 4.13: Store field instrumentation

Before instrumentation:

`putfield #54 = Field java.lang.String.hash(I)`

After instrumentation:

ldc_w #2631 = "0hash"

invokestatic #264=Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.StoreField((Ljava/lang/String;)V)

`putfield #54 = Field java.lang.String.hash(I)`

Listing 4.13 indicates that *StoreField* assertion point involves two lines of opcode that inserted to push label 0hash into the stack and calling StoreField() method in the event recognizer.

4.9.10 Instrument union

Each of union instruction in Table 4.1 is responsible for popping two values from the top of the stack. These popped two values will be pushed as one element after combined them.

The instrumentation is as follows:

Listing 4.14: Union instrumentation

Before instrumentation:

```
iadd
```

After instrumentation:

```
invokestatic #2670 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Union(()V)
iadd
```

Listing 4.14 indicates that *Union* assertion point involves one line of opcode that inserted to call Union() method in the event recognizer.

4.9.11 Instrument new

New includes creating new object (*class*, *array*, or *interface* type). The object reference name will again be prefixed with 0.
The instrumentation is as follows: Example:

Listing 4.15: New instrumentation

Before instrumentation:

```
new #94 = Class java.io.File
```

After instrumentation:

```
ldc_w #393 = "0java.io.File"
invokestatic #395 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.New((Ljava/lang/Object;)V)
new #94 = Class java.io.File
```

Listing 4.15 indicates that *New* assertion point involves two lines of opcode that inserted to push label 0java.io.File into the stack and calling New() method in the event recognizer.

4.9.12 Instrument newarray

Newarray includes three instructions:

- newarray: Create new array.

- anewarray: Create new array of reference.
- multianewarray: Create new multidimensional array.

Newarray instructions pop the count (array size) from the stack and push the arrayref onto the top of the stack.

The instrumentation is as follows:

Listing 4.16: Newarray instrumentation

Before instrumentation:

```
newarray byte
```

After instrumentation:

```
ldc_w #298 = "0NewArray"
invokestatic#301=Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.NewArray((Ljava/lang/String;)V)
newarray byte
```

Listing 4.16 indicates that *NewArray* assertion point involves two lines of opcode that inserted to push label 0NewArray into the stack and calling NewArray() method in the event recognizer.

4.9.13 Instrument monitor

Monitor category includes two instruction *monitorenter* and *monitorexit*. These instructions should be instrumented because they manipulate the stack as follow

- *monitorenter* pushes the monitored object onto the stack.
- *monitorexit* pops the monitored object from the stack.

Thus, these two instructions may change the way that information flow.

The instrumentation is as follows:

Listing 4.17: Monitor instrumentation

Before instrumentation:

```
monitorenter
```

After instrumentation:

invokestatic #1557 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Monitor(()V)

monitorenter

Listing 4.17 indicates that *Monitor* assertion point involves one line of opcode that inserted to call Monitor() method in the event recognizer.

4.9.14 Instrument invoke

There are several different mechanisms for invoking methods in the JVM.

- *Invokevirtual*: invoke instance method; dispatch based on class.
- *Invokespecial*: invoke instance method; special handling for superclass, private, and instance initialization method invocations.
- *Invokestatic*: invoke a class (static) method.
- *Invokeinterface*: invoke interface method.

Let m be the method name. The actual method to be instrumented is selected according to Listing 4.4.

4.9.14.1 Instrument native write

If m is native and named *write* then it will be instrumented as follows.

Listing 4.18: Write native method instrumentation

Before instrumentation:

invokevirtual #110 = Method java.io.OutputStream.write((I)V)

After instrumentation:

iconst_1

iconst_0

```
invokestatic #394 = Method Monitor.EventRecognizer.NativeWrite((II)V)
invokevirtual #110 = Method java.io.OutputStream.write((I)V)
```

Listing 4.18 indicates that *NativeWrite* assertion point involves three lines of opcode that inserted to push constant 1 (parameters) and constant 0 (return values) into the stack and calling NativeWrite() method in the event recognizer.

4.9.14.2 Instrument native method

If *m* is another *native* method then it will be instrumented as follows:

Listing 4.19: Native method instrumentation

```
Before instrumentation:
invokevirtual #30 = Method java.io.Writer.flush(()V)

After instrumentation:
iconst_1
iconst_0
invokestatic#886=Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.NativeMethod((II)V)
invokevirtual #30 = Method java.io.Writer.flush(()V)
```

In the Listing 4.19 the *flush()* method is instrumented as native method with one parameter (*iconst_1*) and 0 return value (*iconst_0*).

4.9.14.3 Instrument method

If *m* is another method then it will be instrumented as follows:

Listing 4.20: Method instrumentation

```
Before instrumentation:
  invokevirtual #45 = Method java.nio.ByteBuffer.position(()I)

After instrumentation:
ldc_w #1560 = "position"
iconst_1
iconst_1
invokestatic #152=Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Method((Ljava/lang/String;II)V)
```

```
invokevirtual #45 = Method java.nio.ByteBuffer.position(()I)
```

As shown in Listing 4.20 the following information about *position()* method is sent to the event recognizer:

- The method name: *position*
- The method parameters number: *iconst_1*
- The number of expected return values: *iconst_1*

All this information is detected from the method signature before instrumentation.

4.9.14.4 Instrument special method

If *m* is constructor then it will be instrumented as follows.

Listing 4.21: Special method instrumentation

Before instrumentation:

```
invokespecial #96 = Method java.io.File.<init>((Ljava/lang/String;)V)
```

After instrumentation:

```
invokestatic #401 =Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.SpecialMethod(()V)
invokespecial #96 = Method java.io.File.<init>((Ljava/lang/String;)V)
```

Listing 4.21 indicates that *SpecialMethod* assertion point involves one line of opcode that inserted to call SpecialMethod() method in the event recognizer.

4.10 Implicit Information Flow Instrumentation.

In Java bytecode, the scope of the implicit information flow occurs within condition or repetitive commands. The stack maybe manipulated in different ways according to the branch of branching instruction offset address that may cause performing a different number of pop and push operations. The following subsections show the

instrumentation of the instructions that deals with the implicit information flow as indicated in the method body instrumentation Listing 4.4.

4.10.1 Instrument ifcond

Ifcond involves all condition instructions that pop one value from the stack as illustrated in Table 4.1.

The instrumentation is as follows:

Listing 4.22: If condition instrumentation

Before instrumentation:

`ifle` 31

After instrumentation:

invokestatic #1141 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ifcond(()V)

`ifle` 43

As shown in List 4.22 that the offset address of the condition *ifle* is addresses to 31. However, after instrumenting all bytecode instructions the offset address may changed according to the size of the inserted assertion points as is illustrated in Listing 4.22 the offset address has changed to 43.

4.10.2 Instrument ifcmp

Ifcmp includes all condition instructions that pop two values from the stack as indicated in Table 4.1.

The instrumentation is as follows:

Listing 4.23: Ifcmp condition instrumentation

Before instrumentation:

`if_icmpeq` 75

After instrumentation:

invokestatic #378 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ifcmp(()V)

`if_icmpeq` 94

4.10.3 Instrument switch

Switch includes compound conditional branch instructions:

- Tableswitch: Access jump table by index and jump.
- Lookupswitch: Access jump table by key match and jump

The instrumentation is as follows:

Listing 4.24: Table switch instrumentation

Before instrumentation:

```
tableswitch {
                default: 34
                0: 28
                1: 30
                2: 32
               }
```

After instrumentation:

```
invokevirtual #39 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ifcond(()V)
tableswitch {
                default: 93
                0: 55
                1: 67
                2: 81
               }
```

4.10.4 Instrument endif

Similar to (Ball 1993), a condition region should be created for each *If* <*cond*> to control implicit information flow. The condition region will be created according to the offset address of the if statement. The start of the region is at the beginning of the *if* statement and the end of the region is at the end of the *if* statement. As indicated in the instrumentation algorithm Listing 4.4 the assertion point of the start of the region is inserted after step 2 as illustrated in Sections 4.10.1 and 4.10.2

respectively. The end of the region is where the *Endif* should be instrumented which is after fourth step of the instrumentation algorithm.

Each type of *ifcmp* and *ifcond* statements in Table 4.1 has its own offset address. *Endif* will be instrumented according to the offset address of the *If* <*cond*> instruction opcode. The offset address must be within the method that contains the *If* <*cond*> instruction. If the opcode instruction before the target opcode index is not a goto statement then the assertion point *Endif* will inserted in the index as specified in the *If* <*cond*> offset address. as illustrated in the examples of Listings 4.25 and 4.26 respectively.

Example 1:

Assume that the *if* statement has a conditional form as illustrated in Figure 4.5. When the condition is true, the statements between the *if* condition and the *endif* will be performed. Listing 4.25 shows the *if* condition original bytecode, the instrumented bytecode after performing step 2 of the method body instrumentation algorithm in Section 4.8.4 and the changes in the bytecode after performing step 4 of the method body instrumentation algorithm after inserting the *Endif* assertion point.

```
if <cond> then
    statement1;
    statement2
        .....
        .....
endif
```

Figure 4.5: Condition statement form example 1

Listing 4.25: Example 1 Endif instrumentation

Before instrumentation:

```
03: ifne 8
06: iconst_1
07: istore_1
08: iconst_2
```

After step 2 of the method body instrumentation `algorithm`:

18: invokestatic #44 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ifcond(()V)

21: `ifne` 35

24: invokestatic #46 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

27: `iconst_1`

28: dup

29: ldc #47 = "1"

31: invokestatic #49 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store((Ljava/lang/Object;
Ljava/lang/String;)V)

34: `istore_1`

35: invokestatic #51 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

38: `iconst_2`

After step 4 of the method body instrumentation `algorithm`:

18: invokestatic #44 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ifcond(()V)

21: `ifne` 38

24: invokestatic #46 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

27: `iconst_1`

28: dup

29: ldc #47 = "1"

31: invokestatic #49 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store((Ljava/lang/Object;
Ljava/lang/String;)V)

34: `istore_1`

35: invokestatic #61 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Endif(()V)

38: invokestatic #51 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

41: `iconst_2`

The *if* condition in the original bytecode Listing 4.25 is started at opcode index 03 and has offset address 8. The end of the *if* condition is at opcode index 8 as illustrated in Figure 4.6 (Before instrumentation). However, after performing step 2 of the method body instrumentation algorithm of Section 4.8.4, all original bytecode are instrumented and the start of the region is inserted in the first iteration of the instrumentation process. The offset address of the *if* condition will be changed according to the instrumented bytecode inside the *if* condition as shows in Listing 4.25 at opcode index 21.

The offset address is changed to 35 as illustrated in Figure 4.6 (after step 2 of the

method body instrumentation algorithm). Finally, step 4 of the method body instrumentation algorithm is performed where the end of the region (*Endif*) is inserted. The offset address of the *if* condition will be modified to point to the opcode index 38. The *Endif* assertion point will be inserted before offset address of the *if* condition as show in Listing 4.25 at opcode index 35. Figure 4.6 illustrates the creation of the region in the instrumentation process.

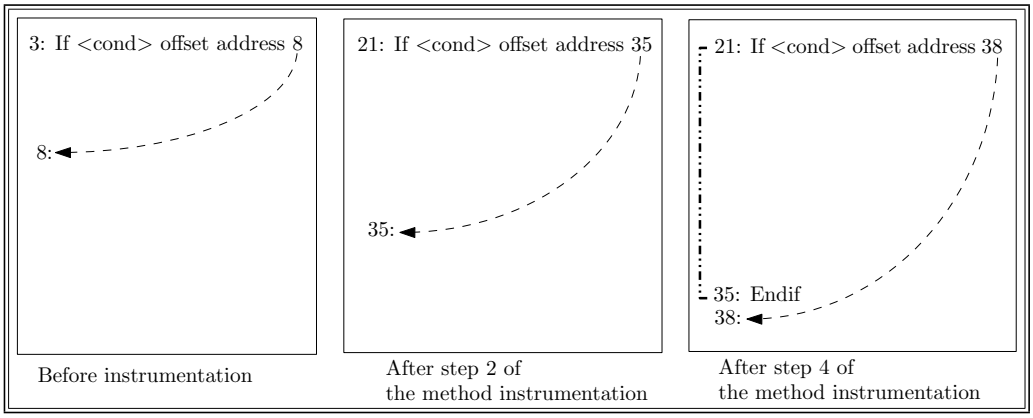


Figure 4.6: Region creation of example 1

Example 2:

Assume that the if statement has a conditional form as illustrated in Figure 4.7.

```
if <cond> then
    statements;
        .....
else
    statements;
        .....
endif
```

Figure 4.7: Condition statement form example 2

When the condition is true, the statements between the *if* condition and *else* will be performed, otherwise the statements between *else* and *endif* will be performed.

Listing 4.25 shows the *if* condition original bytecode, the instrumented bytecode after performing step 2 of the method body instrumentation algorithm of Section 4.8.4 and the bytecode after inserting the *Endif* assertion point. In this form of the condition both cases true or false will cause an information flow, thus the end of the region will be inserted at the end of the *if* condition.

Listing 4.26: Example 2 of Endif instrumentation

Before instrumentation:

03: `ifne` 11

06: `iconst_1`

07: `istore_1`

08: goto 13

11: `iconst_2`

12: `istore_1`

13: `iconst_1`

After step 2 of the method body instrumentation `algorithm`:

18: invokestatic #45 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ifcond(()V)

21: `ifne` 38

24: invokestatic #47 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

27: `iconst_1`

28: dup

29: ldc #48 = "1"

31: invokestatic #50 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store((Ljava/lang/Object;
Ljava/lang/String;)V)

34: `istore_1`

35: goto 49

38: invokestatic #52 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

41: `iconst_2`

42: dup

43: ldc #53 = "1"

45: invokestatic #55 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store((Ljava/lang/Object;
Ljava/lang/String;)V)

48: `istore_1`

49. invokestatic #57 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

52: `iconst_1`

After step 4 of the method body instrumentation `algorithm`:

18: invokestatic #45 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ifcond(()V)

21: `ifne` 38

24: invokestatic #47 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

27: `iconst_1`

28: dup

29: ldc #48 = "1"

31: invokestatic #50 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store((Ljava/lang/Object; Ljava/lang/String;)V)

34: `istore_1`

35: goto 49

38: invokestatic #52 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

41: `iconst_2`

42: dup

43: ldc #53 = "1"

45: invokestatic #55 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store((Ljava/lang/Object; Ljava/lang/String;)V)

48: `istore_1`.

49:invokestatic #69 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Endif(()V)

52: invokestatic #57 = Method uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const(()V)

55: `iconst_1`

In this example the opcode instruction before the target address is a type of goto statement then the assertion point Endif will be inserted at the target address of the goto statement as illustrated in Listing 4.25. The *if* condition in the original bytecode of Listing 4.25 starts at opcode index 03 and has offset address 11. The end of the *if* condition is at opcode index 8 as illustrated in Figure 4.8 (Before instrumentation). However, after performing step 2, all original bytecode is instrumented and the start of the region is inserted in the first iteration of the instrumentation process. The offset address of the *if* condition will be changed according to the size of the instrumented bytecode inside the *if* condition as shown in Listing 4.25 at opcode index 21. The offset address is changed to 38 as illustrated in Figure 4.8 (after step2 of the method instrumentation algorithm). Finally in the fourth step of the method body instrumentation algorithm the second iteration of the method

bytecode is performed to insert the end of the region (*Endif*).

The instrumentation process gets the offset address of the *if* condition which is 38 as specified at opcode index 21. Checks the instruction that just before the index of the offset address which is the instruction goto because the index of the instruction that is before the offset address 38 is opcode index 35. If the checked instruction is of type *goto* as in the example, then the instrumentation process gets the offset address of the *goto* which is 49 and inserts the *Endif* assertion point at the offset address 49. Figure 4.8 illustrates the creation of the region in the instrumentation process for this form of the *if* condition.

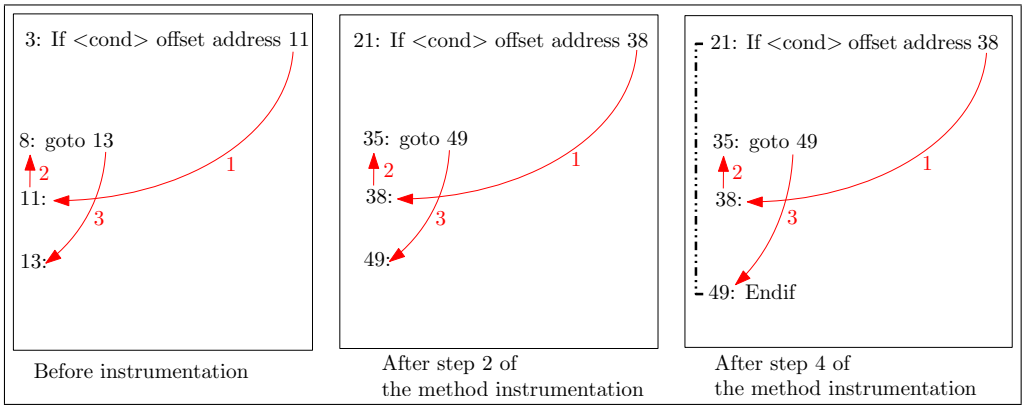


Figure 4.8: Region creation of example 2

However, in case of a return statement inside a region, the return assertion point of Section 4.9.7 has a parameter that holds a counter of the *If* condition. This counter will be increased by 1 each time that *If* condition is performed and decreased by 1 for each performed *Endif*.

4.11 Summary

The presented chapter has introduced the Java bytecode, class loader in Java virtual machine and how a Java class file will be instrumented in order to monitor and control the information flow within a Java application. The chapter provided

example of a Java source file and its bytecode format. This chapter has described the class loader in Java virtual machine and explored the loading phase. The presented chapter has also described the linking process and the initialization process phases. This chapter has discussed a novel instrumentation mechanism for inserting assertion points and provided some examples of Java bytecode instructions before and after instrumentation. Finally, it has discussed the instrumentation of explicit information flow instructions and implicit information flow instructions.

Chapter 5

Runtime Monitoring

Objectives

- Describe how a class file will be executed.
- Describe how information flow will be monitored with respect to an information flow policy.
- Provide the event recognizer and runtime checker algorithms for controlling information flow.

5.1 Introduction

This chapter describes the event recognizer and runtime checker algorithms for controlling information flow within an Java application (the second step of our runtime monitoring mechanism as illustrated in Figure 5.1).

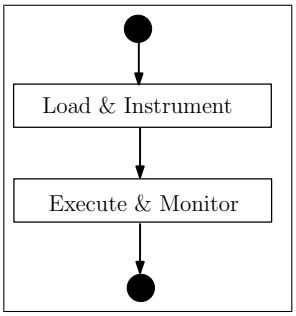


Figure 5.1: Monitoring mechanism flow chart

The chapter is structured as follows. Section 5.2 describes the event recognizer. Section 5.3 provides the explicit information flow algorithms for tracing and controlling explicit information flow. Section 5.4 extends this algorithm for implicit information flow. Section 5.5 describes the runtime checker process. Section 5.6 describes the user feed back component. Finally Section 5.7 describes the information flow policy and its enforcement.

5.2 Event Recognizer

The Java virtual machine is stack oriented, with most operations taking one or more operands from the operand stack of the Java virtual machine's current frame or pushing results back onto the operand stack (Lindholm & Yellin 1997). Our runtime monitoring mechanism is similar as the Java virtual machine runtime frames. In our runtime monitoring mechanism a new runtime frame is created each time a method is invoked. The runtime frame consists of a stack called information flow stack (IFS)

and Symbol_Table for the use by the current method to store its variables. At any point of the execution, there are thus likely to be many frames and equally many information flow stacks (IFS) and Symbol_Tables per method invocation. Only the runtime frame (IFS and Symbol_table) of the current method is active. The event recognizer receives an event that attempts to change the state of the information flow within the application. The event recognizer manipulates all labels of variables using the current runtime frame (IFS and Symbol_table) and implicit information flow stack (IMFS) as illustrated in Figure 5.2.

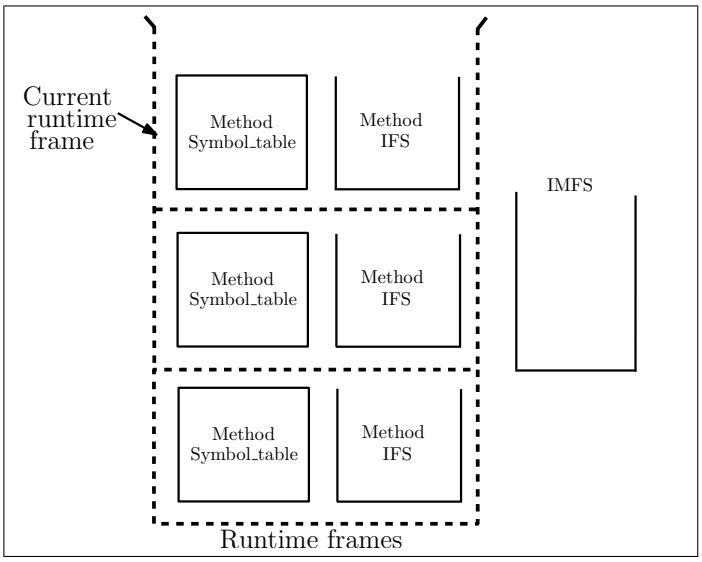


Figure 5.2: runtime frame of the current method and IMFS

5.2.1 Symbol_Tables

An information flow Symbol_Table holds information needed to trace the information flow during runtime. To reduce the time of searching our event recognizer uses a hash table data structure to implement the information flow Symbol_Table as shown in Figure 5.3.

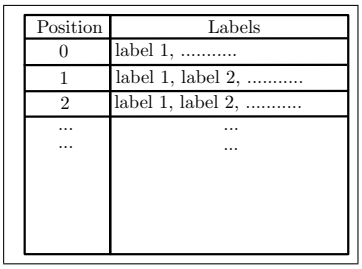


Figure 5.3: Information flow Symbol_Table

As shown in Figure 5.3 the information flow Symbol_Table consists of *positions* and *labels*. The position holds the location of the stored information in the Symbol_Table. Labels are implemented as a set of strings that can hold any number of labels. The event recognizer performs the following operations on the information flow Symbol_Table:

1. Get labels from a specific position.

2. Put labels at a specific position.

5.2.2 Information Flow Stack (IFS)

Each runtime frame contains a last-in-first-out (LIFO) stack known as its information flow stack (IFS). The event recognizer supplies instructions to load labels from Symbol_Tables onto the IFS. The information flow stack is also used to prepare parameters to be passed to other runtime frames and to receive results of other method traces. Our event recognizer uses the information flow stack to control explicit information flow.

5.2.3 Implicit Information Flow Stack (IMFS)

An implicit information flow stack (IMFS) is smiler to the information flow stack (IFS). Our event recognizer uses a shared implicit information flow stack between all runtime frames as illustrated in Figure 5.2. The implicit information flow stack

is shared to control any implicit information flow that may occurs during runtime such as a method invocation inside a conditional statement.

We now move on to the technical details of the event recognizer. The following two Sections 5.3 and 5.4 provide algorithms for inserting assertion points to trace and control information flow.

5.3 Explicit Information Flow Algorithm

As mentioned in Section 4.9 the explicit information flow occurs when using the assignment statement and may occur when using method invocations. The next subsections provide explicit information flow algorithms for all categories of opcodes in Table 4.1 that may cause explicit information flow.

5.3.1 Const Algorithm

The Const assertion point is informing the event recognizer to push the constant label *[Const]* onto the current runtime frame IFS as indicated in the Const algorithm Listing 5.1.

Listing 5.1: Const algorithm

uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Const()

1. **Let x be a List containing the label Const. // [Const]**
2. **IFS.push(x)**.

Figures (5.4a and 5.4b) illustrate the changes in the runtime frame of the current method and IMFS after performing the Const assertion point.

As shown in Figure 5.4b the constant label *[Const]* is pushed onto the top of the current runtime frame IFS.

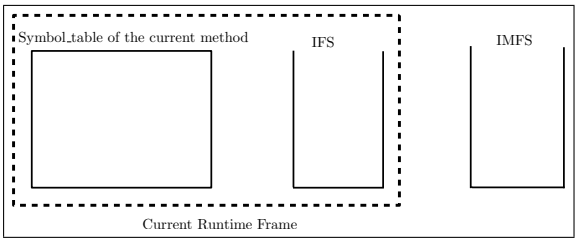


Figure 5.4a: Runtime frame and IMFS before performing the Const assertion point

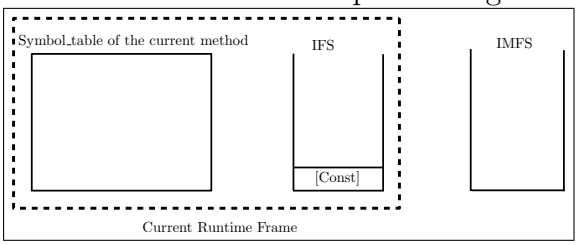


Figure 5.4b: Runtime frame and IMFS after performing the Const assertion point

5.3.2 Load Algorithm

A load assertion point has different types as described in Table 4.1. A load assertion point causes the event recognizer to get a label from the current runtime frame Symbol_Table and push it onto the top of the current runtime frame (IFS) as indicated in Listing 5.2.

Listing 5.2: Load algorithm

uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Load(String keyInSymbolTable)

1. Let temp= Symbol_Table.get(keyInSymbolTable)

2. IFS.push(temp)

Assuming that both current runtime frame IFS and IMFS are empty, the current runtime frame Symbol_Table location 0 contains *Const* and location 1 contains object named *0java,io,File* as illustrated in Figure 5.5a.

The event recognizer receives information that a load operation is going to be performed on location 1. The event recognizer pushes the contents of the Symbol_Table location 1 onto the current runtime frame IFS. Figure 5.5b illustrates the changes in the current runtime frame and IMFS after performing the load asser-

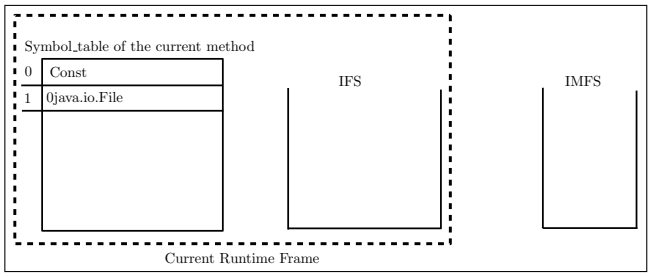


Figure 5.5a: Runtime frame and IMFS before performing the Load assertion point

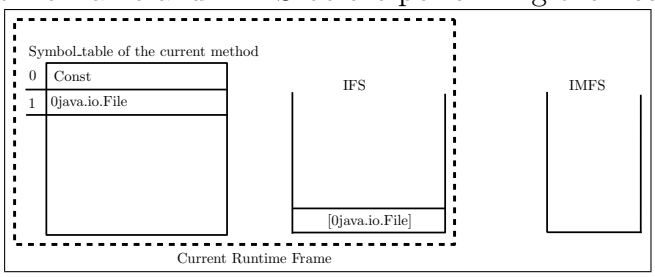


Figure 5.5b: Runtime frame and IMFS after performing the Load assertion point

tion point where the contents of current runtime frame Symbol_Table location are is pushed onto the current runtime frame IFS and there is no change in the IMFS.

5.3.3 Store Algorithm

A store assertion point has different instructions as described in Table 4.1. A store assertion point causes the event recognizer to pop the top element from the current runtime frame IFS and combine it with all contents of the IMFS in one list.Then store the list in current runtime frame Symbol_Table as indicated in Listing 5.3.

Listing 5.3: Store algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Store(String keyInSymbolTable)
  1. Let V and S be an empty lists
  2. V = V U {IFS.pop()}
  3. V = V U IMFS
  5. Symbol_Table.put(KeyInSymbolTable, V).
```

Assuming that the current runtime frame IFS contains one label *[0java.io.File]*, IMFS contains *[/home/secret/file.txt]* and Symbol_Table location 0 has label *Const* and location 1 contains *[0java.io.File]* as illustrated in Figure 5.6a. The event

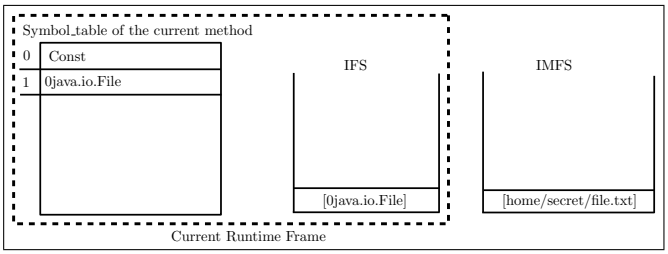


Figure 5.6a: Runtime frame and IMFS before performing the Store assertion point

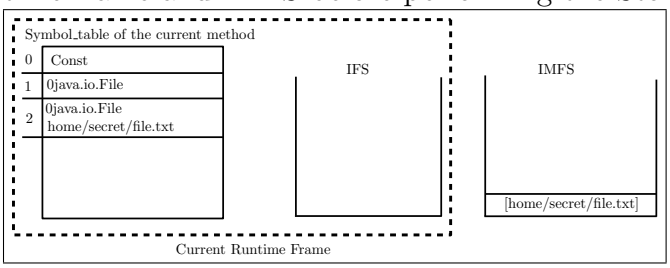


Figure 5.6b: Runtime frame and IMFS after performing the Store assertion point

recognizer receives information that the store operation is going to be performed on location 2. The event recognizer pops the top element from the current runtime frame IFS *0java.io.File* and pops all contents of the IMFS */home/secret/file.txt*. Then the event recognizer combines all popped elements from both stacks in one list *0java,io,File, /home/secret/file.txt*. Finally, it stores the combined elements in the current runtime frame Symbol_Table location 2. Figure 5.6b illustrate the changes in the runtime frame and IMFS after performing the Store assertion point.

5.3.4 Astore Algorithm

The Astore assertion point includes eight different instructions (*iastore, lastore, fastore, dastore, aastore, bastore, castore, sastore*) as indicated in Table 4.1. An Astore assertion point is reporting to the event recognizer that the Astore operation will be the next operation. Then the event recognizer pops the top element from the current runtime frame IFS (*object reference*) and gets the location *Key* of this popped label from the current runtime frame Symbol_Table. It pops the top two elements from the current runtime frame IFS *the value and array index*. It then pops

all contents of the IMFS and combine them with the two popped elements from the current runtime frame IFS into one list. It then stores the list in the current runtime frame Symbol_Table location *key* and pushes all popped elements from the IMFS back in the same order as indicated in Listing 5.4.

Listing 5.4: AStore algorithm

uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Astore()

1. **Let ObjectRef, V and S be empty lists**
2. **ObjectRef = IFS.pop()**
3. **Let Key be a string contains the Objectref location in the current runtime frame Symbol_Table**
4. **Let V = IFS.pop() U IFS.pop**
5. **V = V U IMFS**
6. **Symbol_Table.put(Key, V).**

Figures 5.7a and 5.7b illustrate the current runtime frame and the IMFS before and after performing the Astore assertion point.

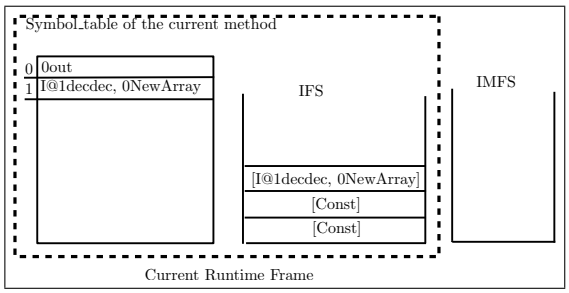


Figure 5.7a: Runtime frame and IMFS before performing the Astore assertion point

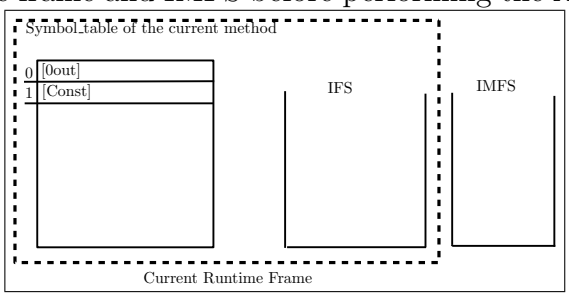


Figure 5.7b: Runtime frame and IMFS after performing the Astore assertion point

As shown in Figure 5.7a the current runtime frame IFS contains three elements the first element is a label to array reference *[I@1decdec, 0NewArray]* and the next

two elements are labels to constants *[Const]* and *[Const]*. The current runtime frame Symbol_Table location 0 contains *0out* and location 1 contains *I@1decdec, 0NewArray*. The event recognizer pops the top element *[I@1decdec, 0NewArray]* from the current runtime frame IFS and get its location 1 from the current runtime frame Symbol_Table. The event recognizer compares the first element in the IFS with all values in the current runtime frame (Symbol_Table) to get the location of the array reference in the Symbol_Table as shown in Figure 5.7a. Then the top two labels *[Const]* and *[Const]* are popped from the current runtime frame IFS and all contents of the IMFS which in our example is empty. All popped elements from both stacks will be combined together and stored in the current runtime frame Symbol_Table location 1 as shown in Figure 5.7b.

5.3.5 Pps Algorithm

The Pps assertion point includes three different instructions as described in Table 4.1. The Pps assertion point has an integer value to specify which type of pps instruction (*pop*, *pop2*, *swap*) will be performed as indicated in the Pps algorithm Listing 5.5.

Listing 5.5: Pps instructions algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Pps(int type)
 if type = 1 // pop
      1. IFS.pop()

else if type = 2 // pop2
        1. x=IFS.pop().
        2. if x is of type (long | double)
           2.1 IFS.pop()

else if type = 3 // Swap
        1. Let x = IFS.pop().
```

```
2. Let y = IFS.pop().
3. IFS.push(x)
4. IFS.push(y)
```

As indicated in Listing 5.5, if the event recognizer receives:

- Type 1, the top element on the current runtime frame IFS will be popped.
- Type 2, the top element on the current runtime frame IFS will be popped and if the type of popped element is long or double then another element will be popped from the current runtime frame IFS.
- Type 3, the two top elements on the current runtime frame IFS will be swapped.

5.3.6 Dup Algorithm

The Dup assertion point includes six different instruction types (*dup*, *dup_x1*, *dup_x2*, *dup2*, *dup2_x1* and *dup2_x2*). The Dup assertion point has two integer values to specify the instruction type as illustrated in the Dup algorithm Listing 5.6.

Listing 5.6: Dup instructions algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Dup(int category, int type) // in
    instrumentation.
if ( category = 1 | (category = 4 & type=1)) // duplicated
    Dup1()
else if ((category = 2) | (category = 5 & type=1)
      | (category = 3 & type= 1) | (category = 6 & type =1)) // dup_x1
    Dup2()
else if ((category = 3 & type= 2) | (category = 6 & type =2)) // dup_x2
    Dup3()
else if category = 4 & type=2 // dup2
    Dup4()
else if category = 5 & type = 2 // dup2_x1
    Dup5()
```

```
else if category = 6 & type = 3 // dup2_x2
    Dup6()
else if category = 6 & type = 4
    Dup7()

Dup1()
    1. Let x = IFS.pop()
    2. IFS.push(x)
    3. IFS.push(x)

Dup2()
    1. Let x = IFS.pop()
    2. Let y =IFS.pop()
    3. IFS.push(x)
    4. IFS.push(y)
    5. IFS.push(x)

Dup3()
    1. Let x = IFS.pop()
    2. Let y = IFS.pop()
    3. Let z = IFS.pop()
    4. IFS.push(x)
    5. IFS.push(y)
    6. IFS.push(z)
    7. IFS.push(x)

Dup4()
    1. Let x = IFS.pop()
    2. Let y = IFS.pop()
    3. IFS.push(x)
    4. IFS.push(y)
    5 IFS.push(x)
    6. IFS.push(y)

Dup5()
    1. Let x = IFS.pop()
    2. Let y = IFS.pop()
    3. Let z = IFS.pop()
    4. IFS.push(x)
    5. IFS.push(y)
    6. IFS.push(z)
```

```
    7. IFS.push(x)
    8. IFS.push(y)

Dup6()
    1. Let x = IFS.pop()
    2. Let y = IFS.pop()
    3. Let z = IFS.pop()
    4. IFS.push(y)
    5. IFS.push(x)
    6. IFS.push(z)
    7. IFS.push(y)
    8. IFS.push(x)

Dup7()
    1. Let x = IFS.pop()
    2. Let y = IFS.pop()
    3. Let z = IFS.pop()
    4. Let w = IFS.pop()
    5. IFS.push(y)
    6. IFS.push(x)
    7. IFS.push(w)
    8. IFS.push(z)
    9. IFS.push(y)
    10.IFS.push(x).
```

As indicated in Listing 5.6, if the event recognizer receives:

- Category 1, the top element on the current runtime frame IFS will be popped, duplicated and the duplicated elements pushed onto the current runtime frame IFS.
- Category 2, duplicate the top element on the current runtime frame IFS and insert the duplicated element two elements down in the current runtime frame IFS.
- Category 3 and type 1, duplicate the top element on the current runtime frame IFS and insert the duplicated element two elements down in the current runtime frame IFS.

- Category 3 and type 2, duplicate the top element on the current runtime frame IFS and insert the duplicated element three elements down in the current runtime frame IFS.
- Category 4 and type 1, duplicate the top element on the current runtime frame IFS and push the duplicated element onto the current runtime frame IFS.
- Category 4 and type 2, duplicate the top two elements on the current runtime frame IFS and push the duplicated elements back onto the current runtime frame IFS in the original order.
- Category 5 and type 1, duplicate the top element on the current runtime frame IFS and insert the duplicated value two elements down in the current runtime frame IFS.
- Category 5 and type 2, duplicate the top two elements on the current runtime frame IFS and insert the duplicated elements in the original order, one element beneath the original elements in the current runtime frame IFS.
- Category 6 and type 1, duplicate the top element on the current runtime frame IFS and insert the duplicated value two elements down in the current runtime frame IFS.
- Category 6 and type 2, duplicate the top element on the current runtime frame IFS and insert the duplicated value three elements down in the current runtime frame IFS.
- Category 6 and type 3, duplicate the top two elements on the current runtime frame IFS and insert the duplicated elements, in the original order, three elements down in the current runtime frame IFS.

- Category 6 and type 4, duplicate the top two elements on the current runtime frame IFS and insert the duplicated elements, in the original order, four elements down in the current runtime frame IFS.

5.3.7 Return Algorithm

A return instruction has twp different types (Return and Treturn). The Return type does not return any value from the current method. The Tretrun type has five different categories (*ireturn*, *lreturn*, *freturn*, *dreturn*, *areturn*) that returns the value from the current method. The event recognizer will perform according to the type of the return assertion point (Return and Treturn) as indicated in Return algorithm Listing 5.7.

Listing 5.7: Return algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Return(IfCounter)
  1. Loop until IFCounter = 0 //IfCounter is if Statement counter of the current method
    1.1 IMFS.pop()
    1.2 IFCounter−−
  2. Destroy the current runtime frame (Current method IFS and Symbol_table)

uk.ac.dmu.msarrab.vif.framework.EventRecognizer.TReturn(IfCounter)
  1. Let ReturnValues, IMFSValues be lists
  2. Loop until IFCounter = 0 //IfCounter is if Statement counter of the current method
    2.1 IMFSValues = IMFS.pop()
    2.2 IFCounter−−
  3. ReturnValues = IFS.pop()
  4. Parent method IFS.push(IMFSValues U ReturnValues).
  5. Destroy the current runtime frame (Current method IFS and Symbol_table)
```

Suppose that, the current runtime frame IFS has one element *0out*, IMFS is empty and the event recognizer receives information from the Treturn assertion point.

The event recognizer pops one element from the current runtime frame IFS, and checks the IMFS whether it has any elements to be popped and combined with

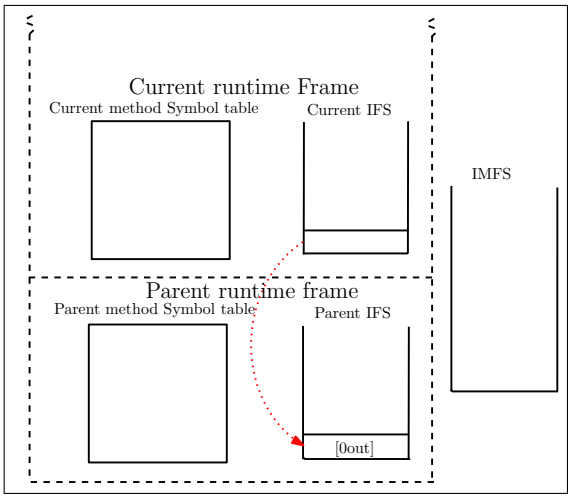


Figure 5.8: Runtime frame and IMFS after performing the TReturn assertion point

the popped element from the current runtime frame IFS. Then the event recognizer pushes all popped elements *[0out]* onto the parent runtime frame IFS. The last step of the Treturn assertion point is to Destroy the current runtime frame (IFS and Symbol_Table) as illustrated in Figure 5.8. The event recognizer pushes the current method return label onto the parent method IFS to trace and control the information flow of the returned labels.

5.3.8 Load Field Algorithm

The load field assertion point has two types (*getstatic* and *getfield*). These report to the event recognizer that a field is about to be loaded. Then the event recognizer pushes the field label onto the current runtime frame IFS.

Listing 5.8: Load field algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.LoadField(String fieldName)
  2. Let V= Symbol_Table.get(fieldName)
  3. If V == null
    3.1 IFS.push(fieldName)
  else
    3.1 IFS.push(V)
```

As indicated in load field algorithm Listing 5.8 that if the type of the loaded field is *getstatic* then the current runtime frame Symbol_Table returns null. Then the event recognizer pushes the field name as received from load field assertion point. Assuming that the current runtime frame Symbol_Table location 0 contains *0out*, location 0fd contains *Const*, both stacks (current runtime frame IFS and IMFS) are empty and the event recognizer receives from the load field assertion point the load field named 0fd. Then the event recognizer gets the 0fd value from the current runtime frame Symbol_Table and pushes the 0fd value onto the current runtime frame IFS as illustrated in Figures 5.9a and 5.9b.

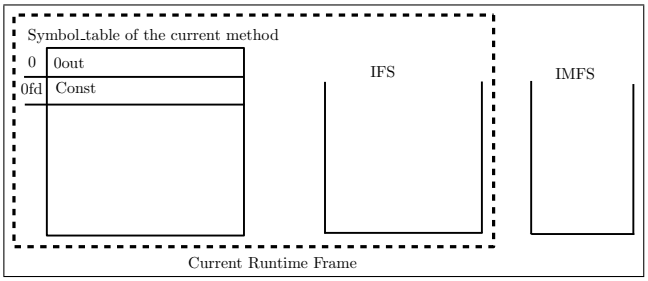


Figure 5.9a: Runtime frame and IMFS before performing the LoadField assertion point

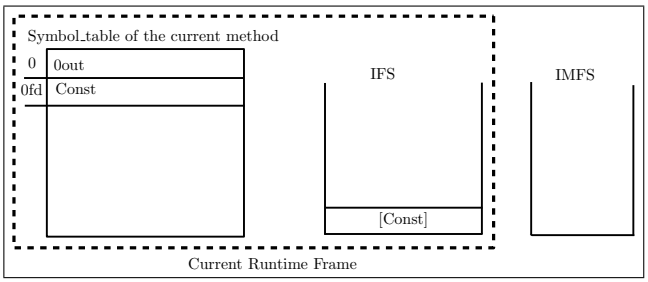


Figure 5.9b: Runtime frame and IMFS after performing the LoadField assertion point

5.3.9 Store Field Algorithm

The store field assertion point has two types of instructions (*putfield* and *putstatic*). The store field algorithm causes the event recognizer to pop the top element from the current runtime frame IFS and all contents of the IMFS. It then combines all

popped elements into one list and stores the list in location FieldName of the current runtime frame Symbol_Table.

Listing 5.9: StoreField algorithm

uk.ac.dmu.msarrab.vif.framework.EventRecognizer.StoreField(String fieldName)

1. **Let Values and S be a lists**
2. **Values = IFS.pop()**
3. **Values = Values U IMFS**
4. **Symbol_Table.put(FieldName, Values).**

Suppose that the current runtime frame IFS has one label *[Const]*, IMFS has two elements the top element is *[Const]* and the bottom element is *[/home/file.txt]* and the current runtime frame Symbol_table has one label *0out* in location 0.
The event recognizer receives information from the store field assertion point that the field named *0fd* is about to be stored. The event recognizer pops the top element from the current runtime frame IFS *Const* and all elements of the IMFS *Const, /home/file.txt*. Then all popped elements will be combined as *Const, /home/file.txt* and stored in the current runtime frame Symbol_table location 0fd as illustrated in Figures 5.10a and 5.10b.

Thus, the popped label from the current runtime frame (IFS) *[Const]* is explicitly flowing to the field named 0fd and all labels in the IMFS (*[Const]* and *[/home/-file.txt]*) are implicitly flowing to the same field 0fd as illustrated in Figure 5.10b.

5.3.10 Union Algorithm

Each of the union instruction family (Table 4.1) is responsible for popping two labels from the top of the current runtime frame IFS. These two labels will be pushed as one element after combining them, independent of their types.

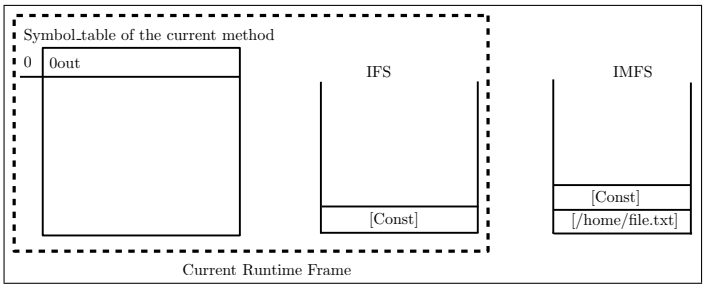


Figure 5.10a: Runtime frame and IMFS before performing the StoreField assertion point

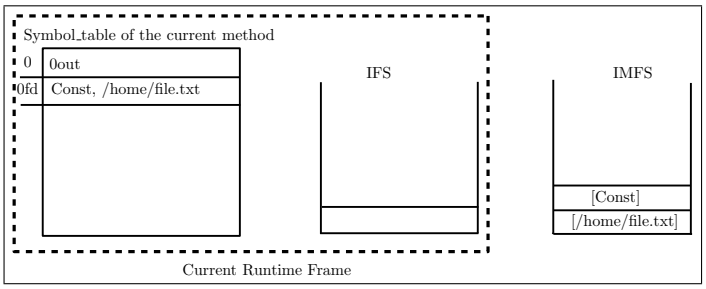


Figure 5.10b: Runtime frame and IMFS after performing the StoreField assertion point

Listing 5.10: Union algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Union()
  1. Let x = IFS.pop()
  2. Let y = IFS.pop()
  3. IFS.push(x U y)
```

Assume that, the current runtime frame IFS has two elements, the top one is a label *[Const]* and the bottom element is a label of static field named *[0out]*. The event recognizer receives information from the union assertion point that one of the union instruction is going to be performed. The event recognizer will pop the two top elements from the current runtime frame IFS. These values are combined and pushed as one element onto the current runtime frame IFS as shown in Figure 5.11a and 5.11b.

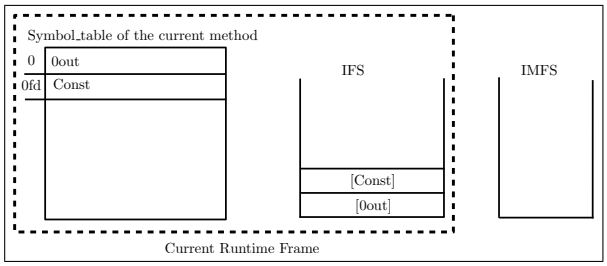


Figure 5.11a: Runtime frame and IMFS before performing the Union assertion point

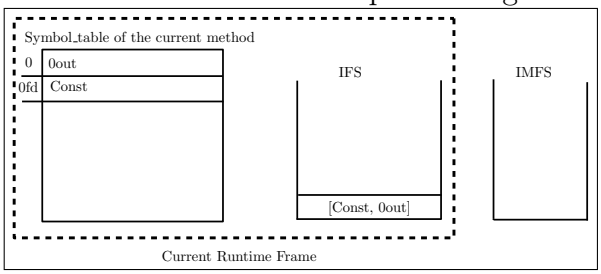


Figure 5.11b: Runtime frame and IMFS after performing the Union assertion point

5.3.11 New Algorithm

The event recognizer receives the new object name from the New assertion point and pushes the object reference name onto the current runtime frame IFS.

Listing 5.11: New object algorithm

uk.ac.dmu.msarrab.vif.framework.EventRecognizer.New(String objectName)

1. `IFS.push(objectName)`

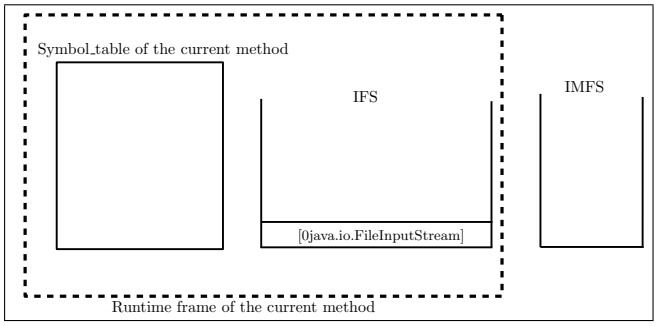


Figure 5.12: Runtime frame and IMFS After performing the New assertion point

The new assertion point will inform the event recognizer that a new object is about to be created. The event recognizer will push the object reference name

[0java.io.FileInputStream] onto the current runtime frame IFS as illustrated in Figure 5.12.

5.3.12 New Array Algorithm

The new array assertion point has three different instructions (*newarray*, *anewarray*, *multianewarray*) as indicated in Table 4.1. The new array assertion point will inform the event recognizer that a new array object is about to be created. The event recognizer pops the top element from the current runtime frame IFS indicating the *array size* and pushes the reserved array object name *0NewArray* onto the current runtime frame IFS as shown in Figure 5.13.

Listing 5.12: NewArray object algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.NewArray(ArrayObjectName)
1. IFS.pop() // Array size
2. IFS.push(ArrayObjectName)
```

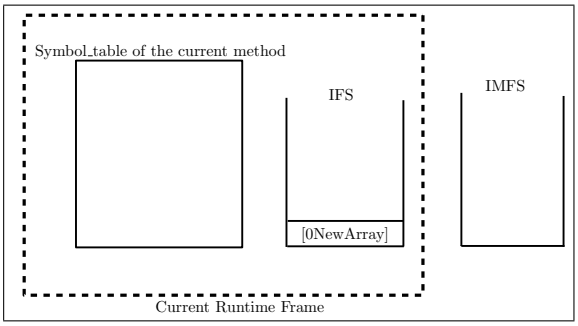


Figure 5.13: Runtime frame and IMFS After performing the New array assertion point

5.3.13 Monitor Algorithm

The monitor assertion point has two different instructions (*monitorenter*, *monitorexit*) as indicated in Table 4.1. The Monitor assertion point is asking the event

recognizer to pop the top element in current runtime frame IFS as indicated in Listing 5.13.

Listing 5.13: Monitor algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Monitor()
  1. x= IFS.pop()
```

Assuming that the current runtime frame IFS has two elements, the top one is a label to object reference *[I@5fec4ec]* and the bottom one is a label to constant *[Const]*. The event recognizer receives information from the monitor assertion point and then the event recognizer pops the top element from the current runtime frame IFS as shown in the Figures 5.14a and 5.14b.

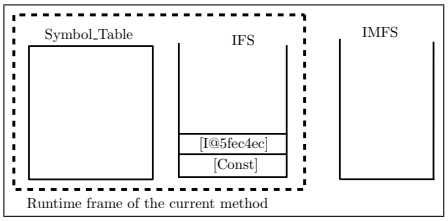


Figure 5.14a: Runtime frame and IMFS before performing the Monitor assertion point

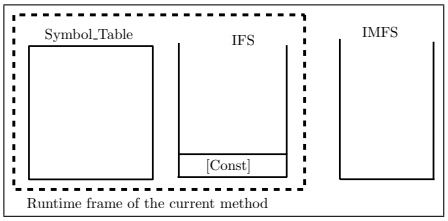


Figure 5.14b: Runtime frame and IMFS after performing the Monitor assertion point

5.3.14 Native Method Algorithm

The Native method is a method that is written in a language other than the Java programming language. The event recognizer deals with the native method assertion point in a way that all its parameters will be combined together as one element and this element will be returned many times according to the number of the native

method return values. The Native method assertion point will report to the event recognizer that the native method is about to be invoked with the parameters and return value numbers. Then the event recognizer pops the top label or labels from the current runtime frame IFS according to the parameters number and combine all popped labels into one list. Then it combines all the contents of the IMFS in the same list. Finally, it pushes the list of the labels onto the current runtime frame IFS many times according the return value numbers as indicated by the Native write method algorithm Listing 5.14.

Listing 5.14: Native method algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.NativeMethod(int parametersNumber, int
    returnValues)
 1. Let L be lists
 2. For parametersNumber
    2.1 L = L U IFS.pop()
 3. L = L U IMFS
 4. For returnValues
    4.1 IFS.push(L)
```

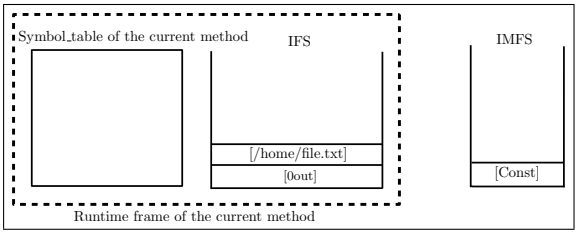


Figure 5.15a: Runtime frame and IMFS before Native method assertion point

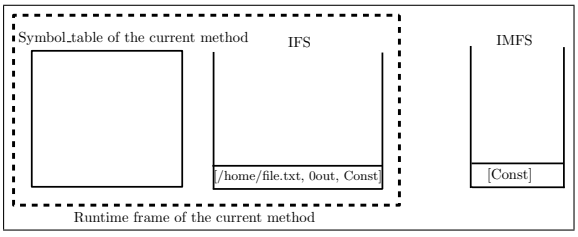


Figure 5.15b: Runtime frame and IMFS after Native method assertion point

As an example suppose that the current runtime frame IFS has two elements. The top element is a label *[/home/file.txt]* and the bottom element is label *[0out]* and

the IMFS has one value *[Const]*. The event recognizer receives information about a native method with two parameters and one return value. The event recognizer pops the top two labels *[/home/file.txt]* and *[0out]* from the current runtime frame IFS and pop all contents of the IMFS *[Const]*. Then all popped elements from both stacks will be combined as one element */home/file.txt, 0out, Const*. The combined values will be pushed onto the current runtime frame IFS one time according to the return value number and the popped elements from the IMFS will be pushed back in the same order as illustrated in Figures 5.15a and 5.15b

5.3.15 Native Write Method Algorithm

The Native write method assertion point will inform the event recognizer that a native write method is about to be invoked with its parameters numbers. The event recognizer deals with the native write method different by than other native methods because it is the method where the runtime monitoring mechanism intercepts because there the information flow may take place. To control this flow the event recognizer pops the label or labels from the current runtime frame IFS according to the method parameters number. It combines all contents of IMFS with all popped labels from the IMFS into one list. Finally it sends the popped labels to the runtime checker as indicated by the Native write method algorithm in Listing 5.15.

Listing 5.15: Native write method algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.NativeWrite(int parametersNumber)
  1. Let source, destination be empty lists
  2. if parametersNumber = 3 then
     2.1 Let source = IFS.pop U IFS.pop U IFS.pop // 3 parameters
  else
     2.1 Let source = IFS.pop // 1 parameter
  3. Let destination = IFS.pop
  4. source = source U IMFS
  5. Let programState = Call uk.ac.dmu.msarrab.vif.framework.RuntimeChecker.Check(source,
       destination)
```

```
6. If programState = false
   6.1 Call uk.ac.dmu.msarrab.vif.framework.EventRecognizer.ExecutionExit()
```

The event recognizer sends two sets of events containing the execution trace to the runtime checker. If the runtime checker checks the received set of events and returns true the execution will continue as normal otherwise the event recognizer will terminate the program execution.

5.3.16 Invoked Method Algorithm

The invoked Method assertion point reports to the event recognizer that a method is about to be invoked with the method name and parameters number. Then the event recognizer pops the top label or labels from the current runtime frame IFS according to the parameters number and combine all popped labels into one list. It combines all contents of the IMFS with each popped label in the list of popped labels from the current runtime frame IFS. Then it stores each element in the list in new created runtime frame (Symbol_Table) as indicated in the invoked method algorithm List 5.16.

Listing 5.16: Invoked method algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Method(String methodName, int
    parametersNumber)
 1. Let L be list
 2. Let i= parametersNumber
 3. for i downto 1
   3.1 L[i]= IMFS U IFS.pop()
 4. Create new runtime frame
 5. Let i=0
 6. for i to parametersNumber
   6.1 New runtime frame Symbol_table.put(i,L[i])
```

Assuming that, the current runtime frame IFS has two elements the top element is a label *[0out]* and the bottom element is a label *[Const]* and the IMFS

is empty. The event recognizer is informed about a method invocation named *java.io.PrintStream.println* with two parameters. Then the event recognizer creates a new runtime frame (new IFS and New Symbol_table) and pops the top two elements from the current runtime frame IFS. Finally, it combines each popped element with the contents of the IMFS and stores them in the new runtime frame Symbol_table as illustrated in Figures 5.16a and 5.16b.

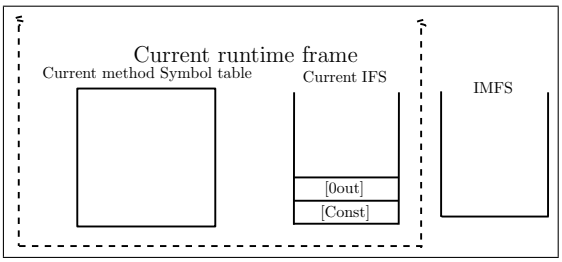


Figure 5.16a: Runtime frame and IMFS before performing the invoked Method assertion point

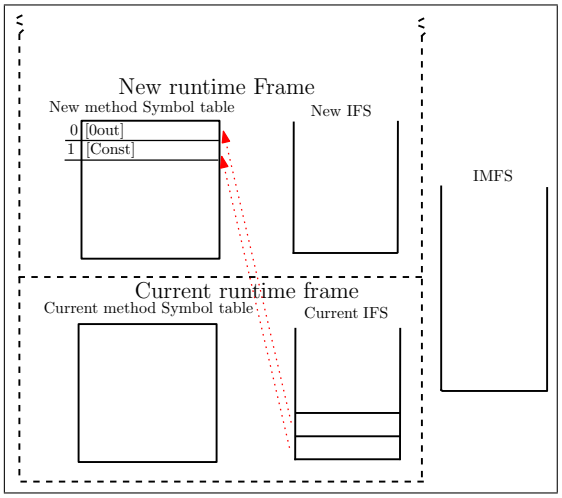


Figure 5.16b: Runtime frame and IMFS after performing the invoked Method assertion point

5.3.17 Special Method Algorithm

The Special method algorithm is used for instance initialization method invocations. Each *invokespecial* method invocation takes three parameters (two arguments val-

ues and objectref). The Special method assertion point will report to the event recognizer that a special method is about to be invoked. Then the event recognizer will pop the top three elements from the current runtime frame IFS and push all popped elements as one element into the current runtime frame IFS as indicated in the Special method algorithm Listing 5.17.

Listing 5.17: Special method algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.SpecialMethod()
  1. Let L be a list
  2. L = IFS.pop() U IFS.pop() U IFS.pop()
  3. IFS.push(L)
```

Suppose that the current runtime frame IFS has three elements, the top value is a label to object reference *[0java.io.File]*, the next label is *[0java.io.File]* and the bottom label is *[/home/file.txt]*.

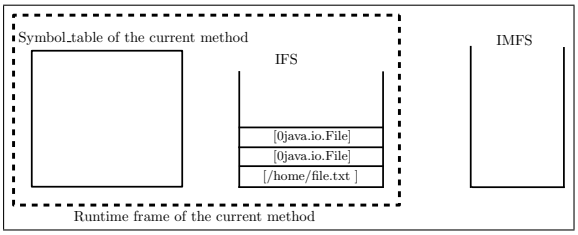


Figure 5.17a: Runtime frame and IMFS before the Special method assertion point

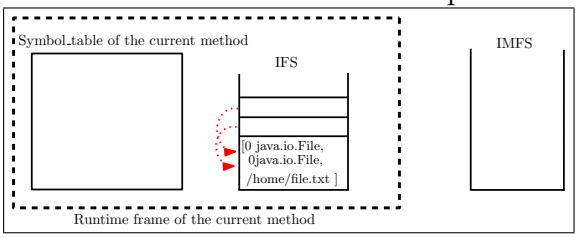


Figure 5.17b: Runtime frame and IMFS after the Special method assertion point

The three elements in the current runtime frame IFS will be popped and pushed onto the current runtime frame IFS as one element *[0java.io.File, 0java.io.File, /home/file.txt]* as shown in Figures 5.17a and 5.17b.

5.4 Implicit Information Flow Algorithm

As mentioned in Section 4.10 implicit information flow occurs within conditional or repetitive commands. The stack maybe manipulated in different ways according to the condition type. Section 5.2.3 described that the event recognizer uses a shared implicit information flow stack between all runtime frames to control implicit information flow. The next subsections show the algorithms for all instructions that deal with the implicit information flow.

5.4.1 Ifcond Algorithm

The Ifcond assertion point has eight different instructions (*ifeq*, *ifne*, *iflt*, *ifge*, *ifgt*, *ifle*, *ifnull*, *ifnonnull*) as indicated in Table 4.1. The Ifcond assertion point causes the event recognizer to pop the top element from the current runtime frame IFS and push the popped element onto the top of IMFS as indicated in the Ifcond algorithm in Listing 5.18.

Listing 5.18: Ifcond algorithm

uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Ifcond()

1. **Let L be a list**
2. **L = IFS.pop()**
3. **IMFS.push(L)**

Suppose that the current runtime frame IFS has two elements, the top element is label *[home/file.txt]* and the bottom element is label *[Const]*. The event recognizer receives an event from the assertion point and Ifcond then the top element in the current runtime frame IFS *[/home/file.txt]* will be popped from the IFS and pushed onto the IMFS as illustrated in Figures (5.18a and 5.18b.

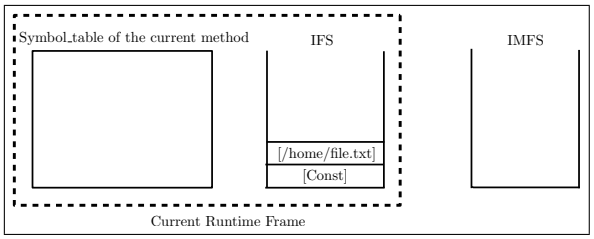


Figure 5.18a: Runtime frame and IMFS before Ifcond assertion point

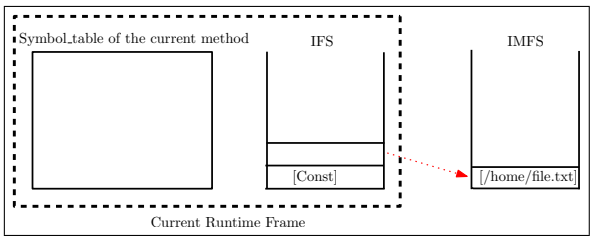


Figure 5.18b: Runtime frame and IMFS after Ifcond assertion point

5.4.2 ifcmp Algorithm

The Ifcmp assertion point includes eight different instructions (*if_icmpeq*, *if_icmpne*, *if_icmplt*, *if_icmpge*, *if_icmpgt*, *if_icmple*, *if_acmpeq*, *if_acmpne*) as indicated in Table 4.1. The Ifcmp assertion point causes the event recognizer to pop the top two labels from current runtime frame IFS and pushes them as one element onto the top of IMFS as indicated in the Ifcmp algorithm in Listing 5.19.

Listing 5.19: Ifcmp algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Ifcmp()
  1. Let L be a list
  2. L = IFS.pop() U IFS.pop()
  3. IMFS.push(L)
```

Assume that the current runtime frame IFS has two elements, the top one is a label *[home/file.txt]* and the bottom one is a label *[Const]*. The event recognizer receives information from the assertion point Ifcmp and then the top two elements in the current runtime frame IFS *[Const]*, *[/home/file.txt]* will be popped and pushed after combining them *[/home/file.txt, Const]* onto the IMFS as illustrated in Figures 5.19a and 5.19b.

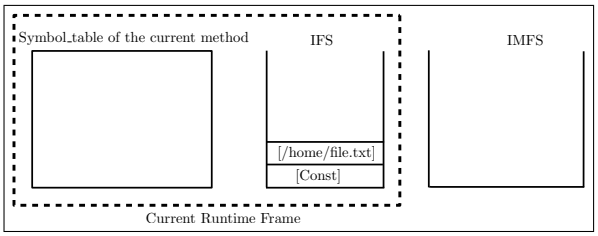


Figure 5.19a: Runtime frame and IMFS before Ifcmp assertion point

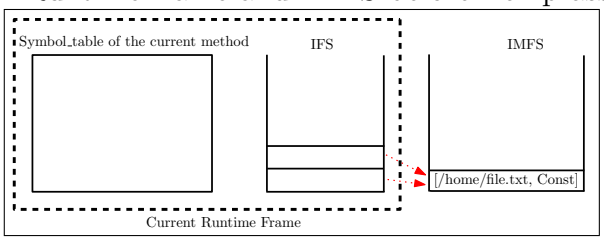


Figure 5.19b: Runtime frame and IMFS after Ifcmp assertion point

5.4.3 Endif Algorithm

The Endif assertion point reports to the event recognizer about the end of the conditional region. Then the event recognizer pops the top element from the IMFS as indicated in the Endif algorithm in Listing 5.20.

Listing 5.20: Endif algorithm

```
uk.ac.dmu.msarrab.vif.framework.EventRecognizer.Endif()
  1. IMFS.pop()
```

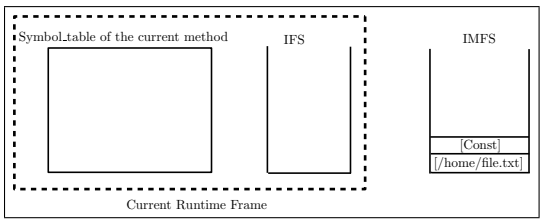


Figure 5.20a: Runtime frame and IMFS before performing the Endif assertion point

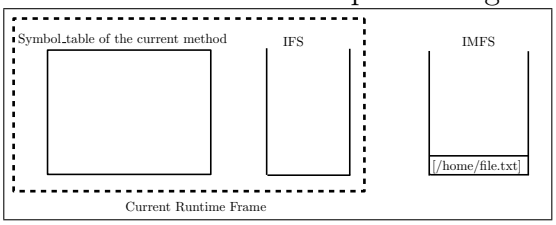


Figure 5.20b: Runtime frame and IMFS after performing the Endif assertion point

Assuming that the IMFS has two elements, the top one is label *[Const]* and the bottom element is label *[/home/file.txt]*. Then the event recognizer will pop the top element as illustrated in Figures 5.20a and 5.20b.

5.5 Runtime Checker

The runtime checker receives events from the event recognizer that may cause information flow within the application. The runtime checker determines whether or not the current events of the execution trace as obtained from the event recognizer satisfies the information flow policy and sends feedback to the user feedback component when it determines that the software is about to enter an insecure state. The runtime checker essentially checks the received set of events that potentially causes information flow. Listing 5.21 presents the events check algorithm.

Listing 5.21: Runtime checker check algorithm

```
uk.ac.dmu.msarrab.vif.framework.RuntimeChecker.Check(list source, list destination)
  1. Let i,j = 0
  2. for i to source.Length
    2.1 Let s = source[i]
    2.2 for j to destination.Length
      2.2.1 Let d = destination[j]
      2.2.2 Let PolicyCheck = uk.ac.dmu.msarrab.vif.framework.InformationFlowPolicy.Check(s, d)
      2.2.3 If PolicyCheck = false
       2.2.4 return true
      else
       2.2.4 uk.ac.dmu.msarrab.vif.framework.Userfeedbackcomponent(s, d)
```

5.6 User Feedback Component

The user feedback component is an interface between our system and the user. An essential functionality of the user feedback component is that all user interaction passes through this component. The user feedback component informs the user

about any feedback received from the runtime checker. As illustrated in our framework Figure 3.2, if the runtime checker determined that this execution would violate the information flow policy then it sends feedback to the user, the system behaviour will be changed accordingly, and the policy will be modified according to the user decision. Chapter 6 describes the user feedback component in detail.

Listing 5.22: User feed back component algorithm

```
uk.ac.dmu.msarrab.vif.framework.Userfeedbackcomponent(List s, List d)
    1. If the user agree about the flow.
     1.1 uk.ac.dmu.msarrab.vif.framework.InformationFlowPolicy.Update(s, d)
     1.2 Return true
    else
     1.1 Return false
```

5.7 Information Flow Policy

An information flow policy is a security policy that defines the authorized paths, which will be a set of rules that regulate how information must flow to prevent leak of information. As described in our framework in Section 3.2, the information flow policy expresses the security requirements as specified by the stakeholder/user to a set of rules that are understandable by our runtime monitoring mechanism. However, it is necessary to detect conflicts in information flow policies. For example if the user defines a new rule it must not conflict with the previous rules. The user must also be able to modify the flow policies during runtime. Policies are dynamic and can be changed in response to the user interaction. Chapter 6 describes our information flow policy in detail.

5.8 Summary

The presented chapter has discussed the second step, a novel runtime monitoring mechanism and shows how the event recognizer and runtime checker deal with the information send by the various assertion points. It has provided different algorithms for tracing and controlling explicit and implicit information flow. Our runtime monitoring mechanism ensures that the program contains only legal flows that are approved by the user. Traditional runtime monitoring only addresses the monitoring of safety properties, i.e. *Functional requirements*. They are not suitable for monitoring information flow or managing the program behaviour at runtime, as there is no feedback from the monitor to the observed software.

Chapter 6

Information Flow Policy and User Interaction

Objectives

- General overview of information flow requirements.
- Describe the information flow policy.
- Explore the user feed back component.

6.1 Introduction

An information flow policy expresses the stakeholders information flow requirements with focus on the different sources and destinations of information that exist within a system and how information may flow between these sources and destinations. In this chapter the focus is on the information flow policy and user feedback component. The remainder of this chapter is structured as follows. Section 6.2 provides a general overview of the information flow requirements. Sections 6.3 and 6.4 describes the information flow policy language. Finally Section 6.5 describes the user feed back component.

6.2 Information Flow Requirements

Information flow requirements are the stakeholders concerns of information flow that come with their system at a high level of abstraction. The main target of the proposed approach is to identify known critical subjects (sources that containing confidential information) and untrusted objects (destinations that should not consume that confidential information). Information flow requirements in our approach are path authorization requirements with constraints on an action that may be performed to allow or deny subjects flowing to an object. In our framework, as described in Section 3.2, in the security requirements specification component, the stakeholders provide the specification of the desired behaviour that a system or subsystem must possess with respect to sensitive information flow. The stakeholders should provide the *sources* of the information and the output channel a *sink* or a *destination* which will be formally expressed in a precise and unambiguous information flow policy. To enable our proposed mechanism to allow or deny the flow of information from source to destination.

6.3 Information Flow Policy

This section is describes how to specify an the information flow policy. An information flow policy expresses the stakeholders requirements in precise and unambiguous form. An information flow policy defines which sources are (dis)allowed to flow to which destinations and in which sources that attempt to flow to specific destinations the user should be asked. An information policy consist of a list of information flow rules.

An **Information Flow Rule** consists of the following three components:

- Action **A**.
- Source **S**.
- Destination **D**

$$\boldsymbol{A\ S \longrightarrow D}$$

Possible actions are + for allowing the flow of the information, **-** disallowing the information flow or **?** for asking the user to allow or disallow the flow of the information.

- A positive information flow form is denoted by $+\ \mathbf{S} \longrightarrow \mathbf{D}$ to (+ denote an action, S a source and D a destination) and is used to explicitly allow information from source S to flow to the destination D. So + is an action that can leak information from source to destination.

- A negative information flow form is denoted by $\mathbf{-\ S} \longrightarrow \mathbf{D}$. Similar to the positive information flow form but it used to explicitly deny or disallow the flow of the information from source S to destination D. So - is an action that can prevent the leak of information from source to destination.

- A user decision form is denoted by **? S** $\longrightarrow$ **D**. Similar to the previous two forms but it is used in case that user should be asked whether the flow of the information from source S to destination D is possible. (?) is an action that can be used to ask the user whether information can flow from source to destination.

Hence, a positive information flow expresses a flow permission and a negative information flow expresses a denial or disallowed flow. Finally, a user decision form flow expresses that decides interactively whether flow is possible or not. An information flow policy consisting of a list of policy rules specifies the restrictions on the possible paths of the information flow.

6.4 Information Flow Policy Language

An information flow policy determines the information flow security measures to be employed within an application to keep the system secure. An information flow policy is a set of policy rules that defines the information flow criteria required to be maintained. The information flow policy works as the reference that controls the flow of the information while the target program is executing.

6.4.1 Syntax

The syntax of the information flow policy language is described in Listing 6.1. The policy definition is introduced by the key word policy and three identifiers.

- <ACTION> which can be (+) positive information flow, (-) negative information flow or user decision (?).
- <ID> is used for the sources S and destinations D names.

Listing 6.1: Information flow policy syntax

```
Policy = (<ACTION> <ID> ">>>" <ID> )

        <ACTION> = "+" | "−" | "?"

        <ID> = <LETTER> (<LETTER> | <CHARACTER>)*>

        <LETTER>= "a"−"z","A"−"Z","0"−"9","_","/"

        <CHARACTER>= ".",":"
```

6.4.2 Semantic of Information Flow Policy Rules

The semantics of an information flow policy defines the possible behaviour of the target program that capture the information flow decisions. An information flow policy rules define the authorized and unauthorized paths of the information flow and when the system user should be asked about the flow decision.

Allowed flow rule example : Assume that

Action A= *+*

Source S = */home/msarrab/secret.txt*

Destination D = *127.1.66.127:2000*

Then the information flow rule is

$$\boxed{+ \ /home/msarrab/secret.txt \longrightarrow 127.1.66.127{:}2000}$$

Hence, information contained in file named */home/msarrab/secret.txt* is allowed to leak to internet socket address *127.1.66.127:2000*.

Disallowed flow rule example : Assume that

Action A= -

Source S = */home/msarrab/secret.txt*

Destination D = *System.out*

Then the information flow rule is.

$$\boxed{\textbf{\textit{- /home/msarrab/secret.txt} } \longrightarrow \textbf{\textit{ System.out}}}$$

Hence, file named */home/msarrab/secret.txt* is not allowed to leak *out* of the running system.

User decision rule example : Assume that

Action A= *?*

Source S = */home/msarrab/secert/msarrab.sec*

Destination D = *127.1.66.177:3000*

Then the information flow rule is.

$$\boxed{\textbf{\textit{? /home/msarrab/secert/msarrab.sec} } \longrightarrow \textbf{\textit{ 127.1.66.177:3000}}}$$

According to the rule the user will be asked whether to allow or disallow the leak of information from the source */home/msarrab/secert/msarrab.sec* to the destination 127.1.66.177:3000.

6.4.3 Information Flow Policy Rules Conflict

The problem of the information flow policy rules conflicts has been addressed using the conflict keyword as indicated in Listing 6.1 Information flow policy syntax. Whereas, in case of a conflict between allow and denial of the flow of the information and ask the user one of the three actions has to have the priority. Listing 6.2 provides the information flow policy with conflicts syntax.

Listing 6.2: Information flow policy with conflicts syntax

```
Policy = (<ACTION> <ID> ">>>" <ID> )* [<CONFLICT>]

        <ACTION> = "+" | "−" | "?"

        <ID> = <LETTER> (<LETTER> | <CHARACTER>)*>

        <LETTER>= "a"−"z","A"−"Z","0"−"9","_","/"

        <CHARACTER>= ".",":"

        <CONFLICT>= "Conflict:" (("+−") ">>>" ("+" | "−" | "?"))
                                ("+? ") ">>>" ("+" | "−" | "?"))
                                ("−? ") ">>>" ("+" | "−" | "?"))
                                ("+−? ") ">>>" ("+" | "−" | "?")))
```

<CONFLICT> is used to address the information flow policy rules conflicts as follows. In the information flow policy four conflict roles can be specified.

- Conflict between allow and denial the flow, +- >>> + denotes allow the flow of the information has the priority.
- Conflict between allow the flow and ask user, +? >>> ? denotes ask user has the priority.
- Conflict between disallow the flow and ask user, -? >>> ? denotes ask user has the priority.
- Conflict between allow, disallow the flow and ask user, +-? >>> ? denotes ask user has the priority.

Hence, an information flow policy defines the possible paths that information can travel securely throughout an application and ultimately between the application and the outside world. In addition, an information flow policy must precisely specify

which action has the priority when there is a conflict between the policy rules. List 6.3 illustrates an example of an information flow policy.

Listing 6.3: Example of information flow policy

```
/* Information flow policy – Written By Mohamed Sarrab, 17/11/10
 */
+/home/msarrab/secret.txt >>> 127.1.66.127:2000
−/home/msarrab/secret.txt >>> System.out
+/home/msarrab/Public/ >>> 162.66.1.123:1000
−/home/msarrab/Public/ >>> System.out
?/home/msarrab/Public/ >>> 162.66.1.177:1000
?/home/msarrab/Public/MSarrab.sec >>> System.out
Conflict: +− >>> +
          +? >>> ?
          −? >>> ?
          +−? >>> ?
```

As indicated in Listing 6.3 the information flow policy has addressed the conflict between two or three actions.

6.5 User Feedback Component

The user feedback component is an interface between a user and the monitored system. An essential functionality of the user feedback component is that all user interaction passes through this component. The user feedback component informs the user about any feedback received from the runtime checker. As illustrated in Figure 3.2 of the framework, if the runtime checker determined that this state execution would violate the information flow policy then it sends feedback to the user through the user feedback component, the system behavior will be changed accordingly, and the information flow policy will be modified according to the user decision.

Assume for example that a program attempts to leak information from source S=*/home/ msarrab/secert/msarrab.sec* to destination D=*127.1.66.177:3000* then the runtime checker will check the information flow policy to figure out if the source

S is allowed or denial to flow this information to destination D. The runtime checker compares all sources in the information flow policy to find any policy rule that has the same source as the present source S= */home/ msarrab/secert/msarrab.sec* and then checks the same rule destination if is it equal to the present destination D= *127.1.66.177:3000* and checks the action of the rule, assuming that the action is (?) as indicated in the next information flow policy rule:

? /home/msarrab/secert/msarrab.sec ⟶ 127.1.66.177:3000

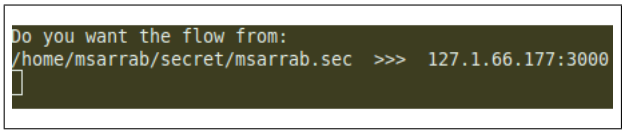


Figure 6.1: A snapshot of monitored flow

According to the action (**?**) of the information flow policy rule the user should be asked as shown in Figure 6.1. The runtime checker sends feedback to the user through the user feedback component where the user made the decision to approve or deny the flow of the information from the source */home/msarrab/secert/msarrab.sec* to the destination *127.1.66.177:3000*.

The user feedback component may receive information from the runtime checker about conflicts in the policy rules, where the conflict rule explicitly stated that user should be asked about the conflict e.g. Listing 6.4 shows an example of information flow policy rule conflicts.

Listing 6.4: Example of information flow policy rule conflicts

```
/* Information flow policy – Written By Mohamed Sarrab, 11/12/10
 */
+/home/msarrab/* >>> 127.1.66.127:2000
−/home/msarrab/private/* >>> 127.1.66.127:2000
Conflict: +− >>> ?
          +? >>> ?
          −? >>> ?
           +−? >>> ?
```

As indicated in Listing 6.4 there is a conflict between two rules. The first rule states that the information can flow from any source in path */home/msarrab/* to destination *127.1.66.127:2000*, while the second rule states that any information in the path */home/msarrab/private/* can not flow to destination *127.1.66.127:2000*. Assuming that the runtime checker received an event from the event recognizer that source named */home/msarrab/private/secret.txt* will flow to destination named *127.1.66.177: 2000*. The runtime checker creates a list to store the actions and checks the information flow policy rules for the source path */home* because the first part of the source path is */home* folder. Then checks the second part of the path */home/msarrab/*, the runtime checker will find that any information from this path can flow (the rule action is +) to destination *127.1.66.127:2000*. Which is applicable to the received source */home/msarrab/private/secret.txt* and destination *127.1.66.177: 2000* from the event recognizer, then the runtime checker stores the action (+) of the current rule in the created list.

However, the runtime checker continues comparing the source */home/msarrab/private/** and the destination *127.1.66.127: 2000* of the second rule. The runtime checker will find that any information from this path */home/msarrab/private/* can not flow (The rule action is -) to destination *127.1.66.127:2000*. Which is applicable to the received source */home/msarrab/private/secret.txt* and destination *127.1.66.177:2000* from the event recognizer, then the runtime checker stores the action (-) of this rule in the list.

Thus, both rules are applicable to the received source and destination. The runtime checker compares the stored actions in the list to find out differences. In this case the actions are different, the first rule action is (+) and the second is (-). The runtime checker checks the conflict rules in the information flow policy for in-conflict any flow rules. However, one of the conflict rules stated that *+- >>> ?* if there is conflict between allow *+* and disallow - then the user will be asked. The user

will receive information through the user feedback component about the conflict as shown in Figure 6.2.

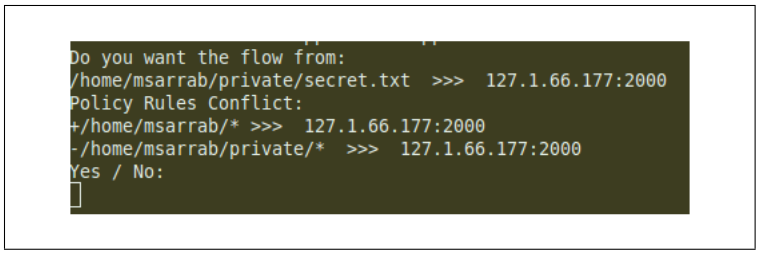


Figure 6.2: A snapshot of user received information (Rules conflict)

The system execution will proceed according to the user decision. If the user responds with Yes, that means the source */home/msarrab/private/secret.txt* is allowed to flow to destination *127.1.66.127:2000* and the execution continues as normal. If the user answers No that means the source */home/msarrab/private/secret.txt* is not allowed to flow to destination *127.1.66.127:2000* and the execution will be terminated.

6.6 Summary

The present chapter introduces the information flow requirements in Section 6.2 where the stakeholders specify the desired strategy and goals about the flow of the information as illustrated in the framework Figure 3.2. Section 6.4 explains the information flow policy language and provides examples of the information flow policy rules. This chapter also addresses the problem of the information flow policy rules conflicts and how the runtime checker, information flow policy and user feedback component interact to monitor in-conflict policy rules. Finally, Section 6.5 explores the user feedback component functionality with more focusing in the user interaction with runtime monitoring mechanism and the user ability to change and modify the program behaviour during runtime. As discussed in Chapter 3 that the only channel

between the user and the monitoring mechanism is the user feedback component. A user interaction with the runtime monitoring provides a flexible security mechanism that can control changeable security requirements.

The main motivation of this proposed approach is that most of the previous research in the information flow control the information flow policy is not in the hand of the end user. The novelty is the user feedback component in which the user interacts with the monitoring mechanism during runtime to manage the program behaviours. One of the key advantage of the user interaction with the monitoring mechanism during runtime is the ability of the user to change program behaviour or modify the way that information flows while the program is executing. An interaction with user provides a flexible and reliable security monitoring mechanism where different users may have different information flow requirements.

Chapter 7

Prototype

Objectives

- Design the prototype of the runtime monitoring mechanism.
- Discuss the implementation of the framework components.
- Support the research that is presented in this thesis.

7.1 Introduction

This chapter provides an introduction to the high level design of the prototype used to control information flow. It also gives a brief introduction to the runtime monitoring mechanism components that are used in the prototype and how they interact with each other to load, instrument and control the flow of the information in the target class file or class files. The rest of this chapter is structured as follows. Section 7.2 introduces which Java library chosen for instrumentation process. Section 7.3 describes the Java agent and its specification. Section 7.4 describes the prototype architecture of the runtime monitoring mechanism and provides the structure of each component. Section 7.5 describes all classes in the system and the static relationships between them. Section 7.6 discusses the execution sequence diagram of runtime monitoring mechanism.

7.2 Java Library for Instrumentation

The development tool selection is the general decision that was taken at prototype development stage. Each bytecode instrumentation library has a different way of working (Chiba & Nishizawa 2003, Aarniala 2005). The selection of the instrumentation library can have a considerable impact on the design and implementation of the whole runtime monitoring mechanism. Three main libraries were compared (BCEL, ASM and Javassist). However after considering for each library the advantages and disadvantages (Joshi 2009), Javassist was chosen as the most suitable library to achieve our approach objectives.

7.3 Java Agent Specification

Java agent is deployed as a JAR file. An attribute in the agent file manifest specifies the agent target class which must be loaded to start the agent. Our framework uses dynamic instrumentation mechanism (Chapter 4) where all classes that are actually loaded will be instrumented including the Java core classes. The class loader (Section 4.5.2) can only provide the classes it defines, not the classes that are delegated to other classes loader thus our approach uses a Java agent to allow all loaded class files (bytecode) to be instrumented and redefined during runtime on one Java virtual machine. In our framework the Java agent is started by specifying an option on the command line. Implementations may also support a mechanism to start agents some time after the JVM has been started. For example, Listing 7.1 shows the command line interface for starting our runtime monitoring mechanism.

Listing 7.1: Command line interface

```
:~$java -javaagent:JavaAgent.jar uk.ac.dmu.msarrab.vif.testtargets.Welcome
```

As illustrated in Listing 7.1 on the implementation an agent is started by adding this option *-javaagent:JavaAgent.jar* to the command line. The *JavaAgent.jar* is the name of the agent. An agent JAR file must conform to the JAR file specification (Oracle 2010). After the Java Virtual Machine is initialized, the premain method will be invoked in the order the agents were specified, then the target application main method will be invoked.

Listing 7.2: Manifest

```
Manifest-Version: 1.0
Premain-Class: Agent.JavaAgent
Boot-Class-Path: /home/msarrab/workspace/UK.AC.DMU.MSARRAB.RVIF/javassist.jar /
    home/msarrab/desktop/JavaAgent.jar
Can-Retransform-Classes: true
Can-Redefine-Classes: true
```

As illustrated in Listing 7.2 the following manifest attributes are defined for an agent JAR file *JavaAgent.jar*:

Premain-Class

The manifest of the agent JAR file in Listing 7.2 must contain the attribute *Premain-Class*. In our framework this is *Premain-Class: Agent.JavaAgent*. The agent class implements a public static premain method similar in principle to the main application entry point.

Boot-Class-Path

A list of paths to be searched by the bootstrap class loader. Paths represent directories or libraries that can be referred to as JAR on many platforms. These paths are searched by the bootstrap class loader after the platform specific mechanisms of locating a class have failed. Paths are searched in the order listed. Paths in the list are separated by one or more spaces. A path takes the syntax of the path component of a hierarchical URI as shown in Listing 7.2.

Boot-Class-Path:

/home/msarrab/workspace/UK.AC.DMU.MSARRAB.RVIF/javassist.jar

/home/msarrab/desktop/JavaAgent.jar

Whereas the first path is to *javassist.jar* library and the second path is to *JavaAgent.jar*.

Can-Redefine-Classes

Can-Redefine-Classes is the ability to redefine classes needed by this agent. Values other than true are considered false.

Can-Retransform-Classes

Can-Retransform-Classes is the ability to retransform classes needed by this agent. Values other than true are considered false.

In order to instrument all loaded classes our runtime monitoring approach uses Java agent to allow all loaded classes (bytecode) to be instrumented and redefined during

runtime on the Java virtual machine.

7.4 Prototype Architecture

As depicted by the Figure 7.1, the runtime monitoring mechanism prototype has been divided into seven major components to fulfil the requirements for controlling the flow of the information within a single Java program as specified in the Section 6.2. It must be noted that none of these components is standalone and all components need to work and interact with each other to achieve the proposed approach goals as indicated in Section 1.4.

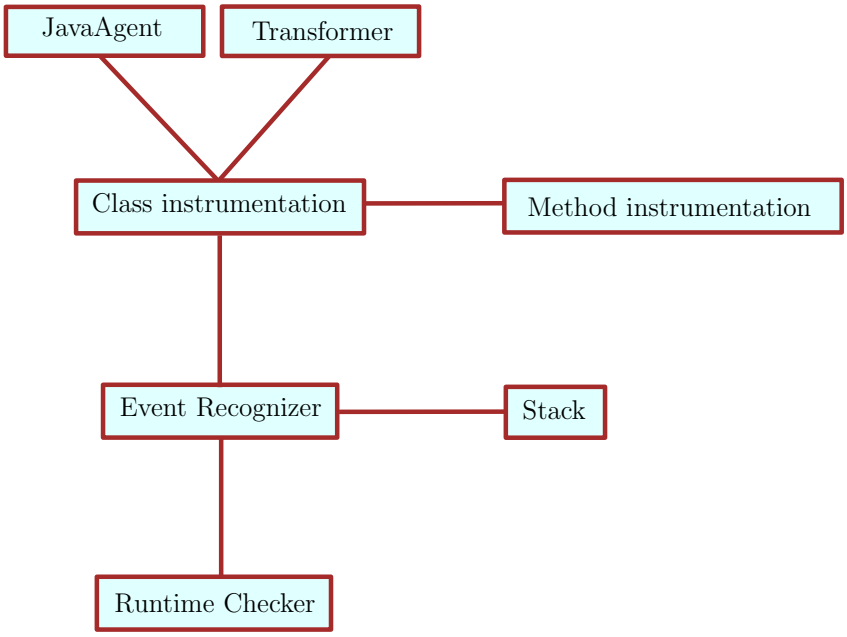


Figure 7.1: Prototype classes

7.4.1 JavaAgent

Agent.JavaAgent

The main aim of this class is to provide services that allow a Java class file to be instrumented during runtime on the Java Virtual Machine as mentioned in Section

7.3. This class has two methods *premain* and *redefineClass*. The *premain* method is similar in principle to the main method application entry point. After the Java Virtual Machine has initialized the *premain* method it allows our runtime monitoring mechanism to instrument all loaded classes before the target class file is actually loaded. The *redefineClass* method is used to redefine all instrumented classes during runtime. At this stage of our runtime monitoring mechanism the Java agent component should perform the following tasks.

- Get all loaded class files.
- Send all loaded class files to be instrumented.
- Redefine all instrumented classes.

Figure 7.2 describes the *JavaAgent* class structure.

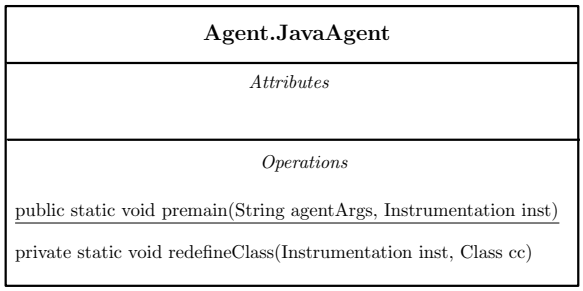


Figure 7.2: Structure of JavaAgent.java

7.4.2 Transformer

Agent.MyTransformer

This class provides services to transform all instrumented class files. The class file transformation occurs before the class is defined by the Java virtual machine. This class controls all loaded classes after the main class file is loaded. This class has one method named *transformer*. The main aim of this method is to transform the supplied class file and return a new replacement class file. This should be done

after the transformer is registered with addTransformer. The transformer will be invoked for all new class definitions and all class redefinitions. Figure 7.3 shows the *MyTransformer* class structure.

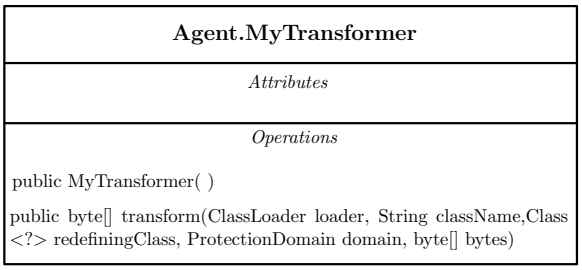


Figure 7.3: Structure of MyTransformer.java

7.4.3 Class Instrumentation

Uk.ac.dmu.msarrab.vif.framework.InstrumentClass

This class aims to instrument the given class file's bytecode such that the instrumented class file produces the required trace information in the given class file. As illustrated in Figure 7.4 this class has one method named *instrument*. The main aim of this method is to send the bytecode of each method in the given class file to the method body instrumentation.

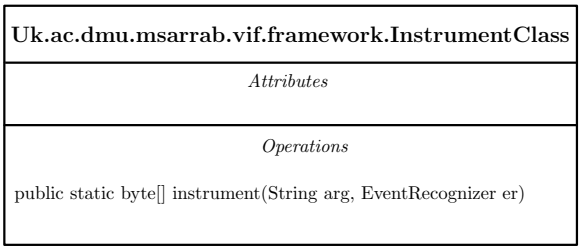


Figure 7.4: Structure of InstrumentClass.java

7.4.4 Method Instrumentation

Uk.ac.dmu.msarrab.vif.framework.InstrumentMethod

The main aim of this class is to instrument all instructions of the received method

bytecode according to the method body instrumentation algorithm Listing 4.4 and as specified in the instruction categories Table 4.1. This class has two methods *methodInstrument* and *methodPrameters* as shown in Figure 7.5. The main aim of the *methodInstrument* is to iterate through the code attribute of a given method and instrument all bytecode instructions based on the instruction's pushes onto or pops from the method stack. The *methodPrameters* method receives the invoked method signature and returns the parameters number of the invoked method.

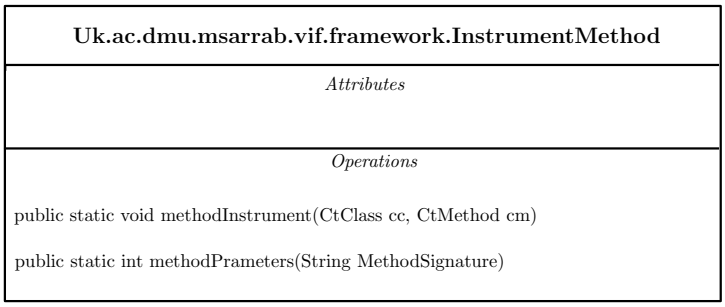


Figure 7.5: Structure of InstrumentMethod.java

7.4.5 Event Recognizer

Uk.ac.dmu.msarrab.vif.framework.EventRecognizer

This class is responsible for manipulating all variables and their values using the explicit information flow stack, the implicit information flow stack and the symbol table as well as creating a new runtime frame for each invoked method and destroy it when that method returns. As mentioned in Section 5.2 the event recognizer uses the current runtime frame (IFS and Symbol_table) and implicit information flow stack (IMFS) to trace the current variables change and uses shared implicit information flow stack (IMFS) to control the implicit information flow stack. This class has a number of methods as shown in Figure 7.6. Each method in this class is responsible for pushing values onto or popping values from the explicit/implicit information flow stack and store or get data from the symbol table. *Method* method

is responsible for creating a new runtime frame (explicit information flow stack and symbol table) and calling the *Return* method will destroy the runtime frame. All other methods in this class manipulating variables and their values according to each method algorithm as described in the explicit information flow algorithms in Section 5.3 and implicit information flow algorithms in Section 5.4.

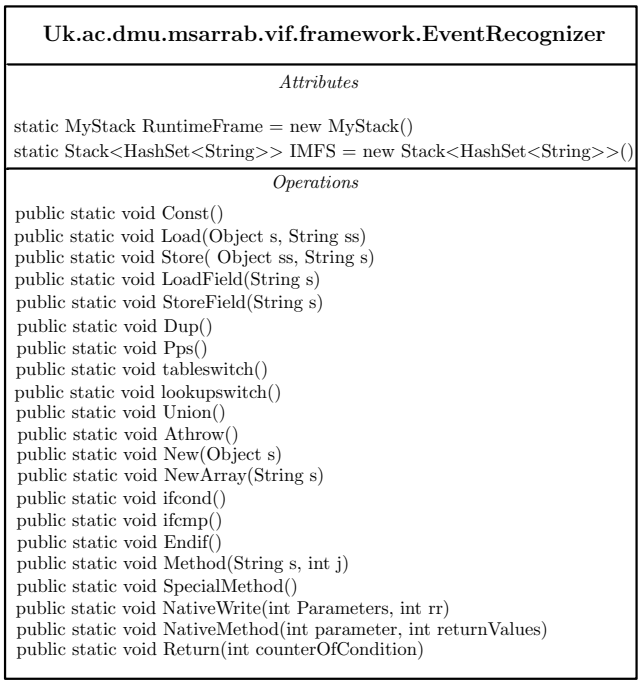


Figure 7.6: Structure of EventRecognizer.java

7.4.6 Stack

Uk.ac.dmu.msarrab.vif.framework.MyStack

The main aim of this class is to trace the flow of the information using runtime frames (Explicit information flow stack and Symbol table) and implicit information flow stack as mentioned in Section 5.2. This class has a number of methods (*openframe*, *currentframe*, *closeframe*, *push*, *pop*, *peek*, *get*, *put* and *tostring*) as shown in Figure 7.7.

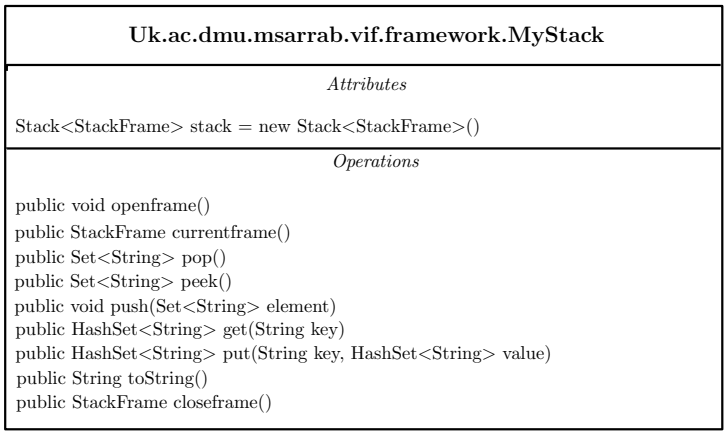


Figure 7.7: Structure of MyStack.java

7.4.7 Runtime Checker

Uk.ac.dmu.msarrab.vif.framework.RunTimeChecker

This class receives the state information from the event recognizer class to check whether or not the current execution trace as obtained from the event recognizer satisfies the information flow policy. This class also sends feedback to the user when it determines that the application is about to enter an insecure state as mentioned in Section 5.5. The Runtime checker class essentially checks the received set of information that potentially causes information flow according to the runtime checker algorithm shown in Listing 5.21. This class has two methods *PolicyCheck* and *AskUser*. The *PolicyCheck* method is checking the potential information flow against a set of rules that are defined in the information flow policy to regulate the flow of the information within the application. The *AskUser* method provides the user with a set of information flow according to the runtime checker algorithm in Section 5.21. Appindex C shows the source code of the runtime checker class. Figure 7.8 shows the *RunTimeChecker* class structure.

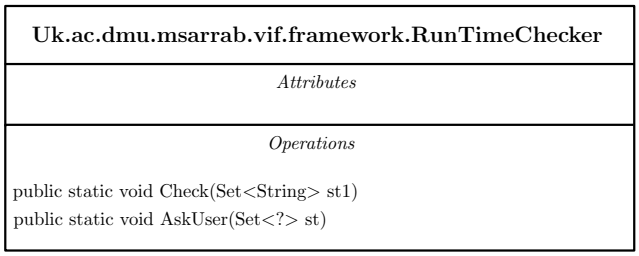


Figure 7.8: Structure of RunTimeChecker.java

7.5 Prototype Class Diagram

As shown in Section 7.4 that each class structure is made up of attributes and operations. Where, attributes define the available information that each class will know about itself and operations are the available processes that a class can perform. These processes in class are called methods. The class diagram in Figure 7.9 describes all classes in the system and the static relationships between these classes. Table 7.1 lists all used classes in the prototype and their related component in the proposed framework shown in Figure 3.2.

Class	Related Component
Agent.JavaAgent	Assertion points
Agent.MyTransformer	Assertion points
Uk.ac.dmu.msarrab.vif.framework.InstrumentClass	Assertion points
Uk.ac.dmu.msarrab.vif.framework.InstrumentMethod	Assertion points
Uk.ac.dmu.msarrab.vif.framework.MyStack	Event Recognizer
Uk.ac.dmu.msarrab.vif.framework.EventRecognizer	Event Recognizer
Uk.ac.dmu.msarrab.vif.framework.RunTimeChecker	Runtime Checker Information flow policy User feedback

Table 7.1: Class and related component

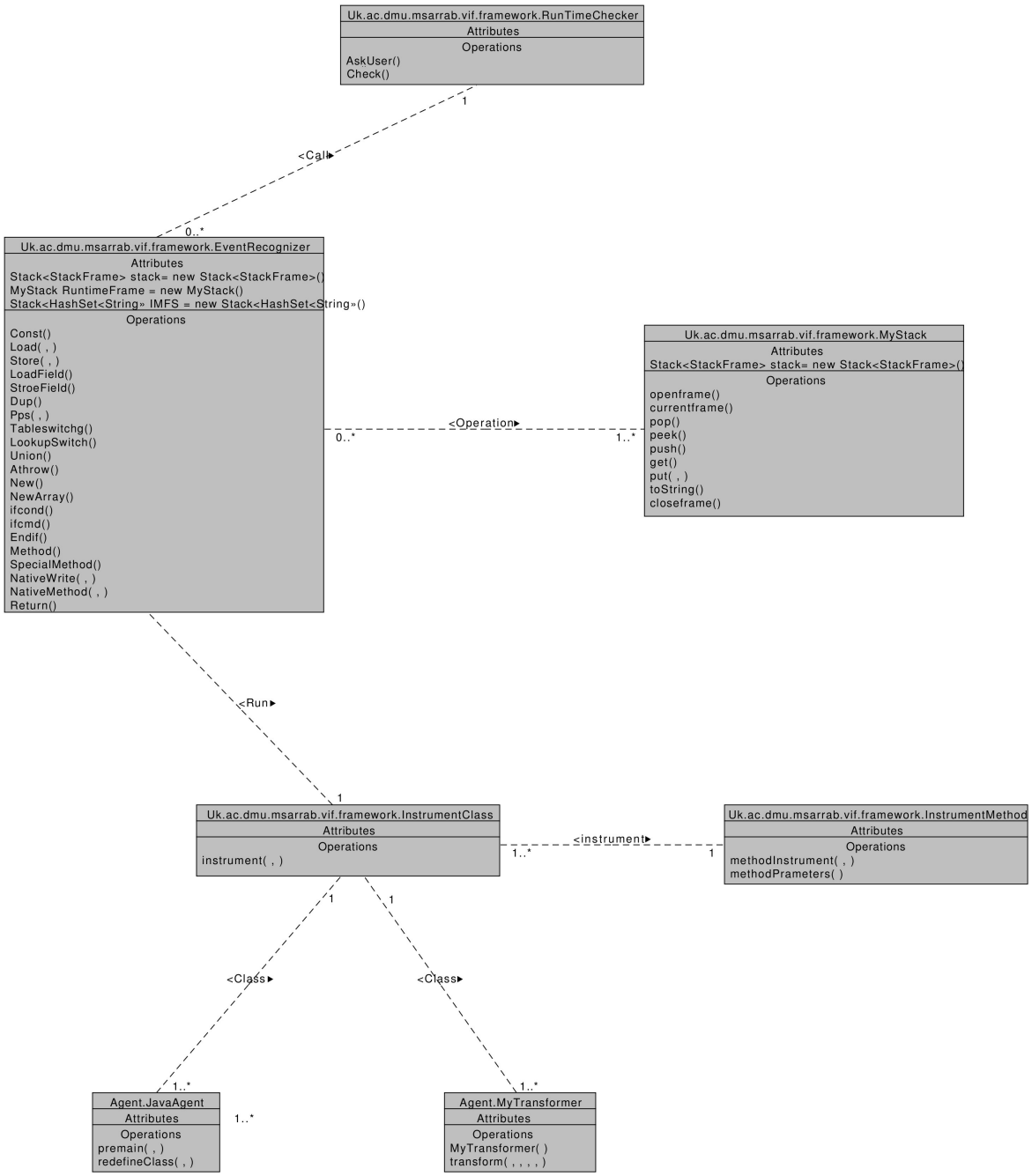


Figure 7.9: Prototype class diagram

7.6 Sequence Diagram

Figure 7.10 illustrates the sequence diagram of execution for the runtime monitoring mechanism. This diagram is based on the prototype implementation of runtime monitoring mechanism.

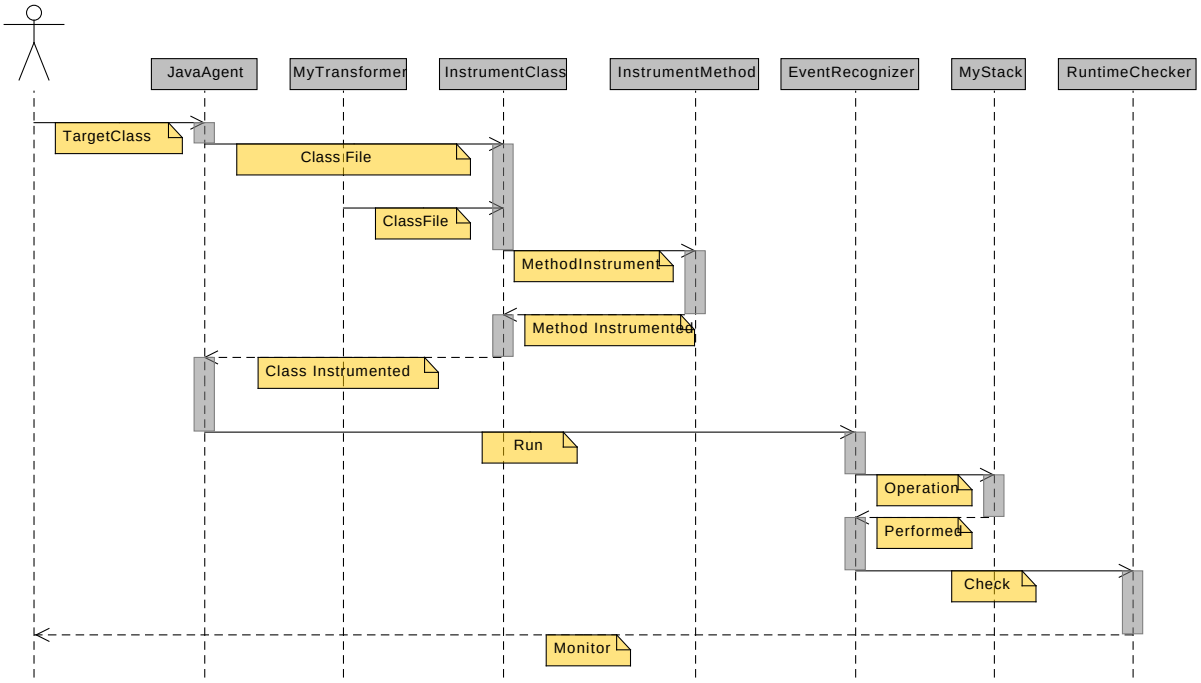


Figure 7.10: Prototype sequence diagram

As shown in Figure 7.10 the sequence diagram is a diagram that depicts the interactions among the runtime monitoring mechanism components, including system participating objects and actors in order to perform the required task.

7.7 Summary

This chapter has presented and discussed the fundamentals of the prototype implementation. This chapter has introduced a high level design of the prototype developed for controlling information flow. It also provided an introduction to the runtime monitoring mechanism components that are used in the prototype and how they interact with each other to load, instrument and control the flow of the information in the target class files. It also described the runtime monitoring mechanism class diagram and sequence diagram.

Chapter 8

Case Studies

Objectives

- Give appropriate case study that shows how a Java program will be traced.
- Give appropriate case study of file sharing system showing how the information flow will be controlled.
- Demonstrate the need and practical applicability of the presented research.

8.1 Introduction

This chapter provides two case studies to illustrate the practical applicability of the presented research. The first case study is a small *helloWorld* program to demonstrate the work of the instrumentation process as described in Chapter 4 and 5. The second case study is a file sharing application to show how the contents of the file will be traced while transfered between a client and a server program. In both case studies the Java source code, the original bytecode and the instrumented bytecode will be provided to show how the flow of the information will be traced and controlled.

8.2 Case Study 1

The presented case study is a small *helloWorld* Java program to demonstrate the interaction of instrumentation process, event recognizer and runtime checker components in more detail. Listing 8.1 shows that the case study consists of one Java class named *helloWorld.java*.

Listing 8.1: Source code of helloWorld.java

```
1 public class Test {
2         public static void main(String[] args) {
3                 System.out.print("hello world");
4         }
5 }
```

The given program in Listing 8.1 has only one method named *helloWorld.main()*. The class execution starts at the main method's first line. Statements are executed one at a time, as ordered in the main method, until the end of the method or another method invocation as shown at line 3 in our example *java.io.PrintStream.print()* method. Table 8.1 lists the execution flow of class *helloWorld.java*.

Table 8.1: Execution flow of class helloWorld.java

```
helloWorld.main()
⟶java.io.PrintStream.print()
    ⟶java.io.PrintStream.write()
        ⟶java.io.PrintStream.ensureOpen()
        ⟶java.io.Writer.write()
            ⟶java.lang.String.length()
            ⟶java.io.Writer.write()
                ⟶java.lang.String.getChars()
                    ⟶java.lang.System.arraycopy()
                ⟶java.io.Writer.write()
        ⟶java.io.BufferedWriter.flushBuffer()
            ⟶java.io.BufferedWriter.ensureOpen()
            ⟶java.io.Writer.write()
        ⟶java.io.OutputStreamWriter.flushBuffer()
            ⟶sun.nio.cs.StreamEncoder.flushBuffer()
                ⟶sun.nio.cs.StreamEncoder.isOpen()
                ⟶sun.nio.cs.StreamEncoder.implFlushBuffer()
                    ⟶sun.nio.cs.StreamEncoder.writeBytes()
                        ⟶java.nio.channels.WritableByteChannel.write()
                        ⟶java.io.OutputStream.write()
                            ⟶java.io.OutputStream.write()
                        ⟶java.nio.Buffer.clear()
                            ⟶java.nio.Buffer.clearMark()
        ⟶java.lang.String.indexOf()
            ⟶java.lang.String.indexOf()
        ⟶java.io.OutputStream.flush()
```

Invoking the built-in method *java.io.PrintStream.print()*, causes another detour of the execution flow to the *java.io.PrintStream.write()* method and when the *java.io.PrintStream.write()* method completes, it picks up where it left off in *java.io.PrintStream.print()*. Finally, it gets back to *helloWorld.main()* until the last statement in the program and then terminates. Technically, the *helloWorld.java* does not terminate yet at the end of the *helloWorld.main()* because the Java interpreter takes care of cleanup of all created objects and then the execution terminates. As discussed in Section 4.8 at start up all loaded classes will be instrumented to trace the execution flow. Table 8.1 lists only the methods used in the execution of *helloWorld.main()*. There are about 200 Java classes that are instrumented during the loading phase. A snapshot of our example execution flow has been taken to show the generated and instrumented bytecode of the first two methods (*helloWorld.main*, *java.io.PrintStream.print()*) and the method that our monitoring mechanism intercepts in *java.io.Writer.write()*. All the loaded classes original and instrumented bytecode can be found in Appendix D. Listing 8.2 presents the generated bytecode of the method *helloWorld.main*.

Listing 8.2: Original bytecode code of helloWorld.java

```
0: getstatic #16 = Field java.lang.System.out(Ljava/io/PrintStream;)
3: ldc #22 = "hello world"
5: invokevirtual #24 = Method java.io.PrintStream.print((Ljava/lang/String;)V)
8: return
```

List 8.3 shows the instrumented bytecode of the method *helloWorld.main*.

Listing 8.3: Instrumented bytecode of helloWorld.java

```
00: ldc #35 = "0out"
02: invokestatic #40 = Method Monitor.EventRecognizer.LoadField((Ljava/lang/String;)V)
05: getstatic #16 = Field java.lang.System.out(Ljava/io/PrintStream;)
08: invokestatic #43 = Method Monitor.EventRecognizer.Const(()V)
11: ldc #22 = "hello world"
```

```
13: ldc #44 = "print"
15: iconst_1
16: iconst_0
17: invokestatic #48 = Method Monitor.EventRecognizer.Method((Ljava/lang/String;II)V)
20: invokevirtual #24 = Method java.io.PrintStream.print((Ljava/lang/String;)V)
23: iconst_0
24: invokestatic #52 = Method Monitor.EventRecognizer.Return((I)V)
27: return
```

After instrumenting all loaded classes one gets bytecode where assertion points are inserted. The instrumented classes are ready to execute. When *helloWorld.main()* start executing, the event recognizer of the runtime monitoring mechanism in Section 5.2 creates a new implicit flow stack (IMFS) and a new runtime frame consisting of an information flow stack (IFS) and a Symbol_able as illustrated in Figure 8.1.

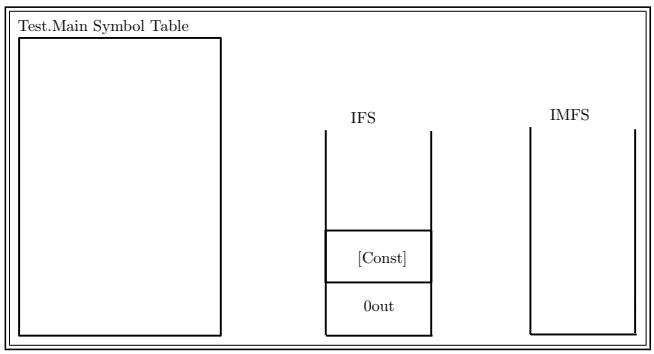


Figure 8.1: The helloWorld.main runtime frame and IMFS

As depicted in Listing 8.3 Opcode index 02 sends an event to the event recognizer to load field named *0out*. The event recognizer pushes *0out* onto the IFS. The second event is at Opcode index 08 that sends a load constant. The event recognizer pushes an empty string onto the top of the IFS. The third event is at Opcode index 17 that informs the event recognizer that another method is about to be invoked with the name *print*, 1 parameter and 0 return value as specified at Opcode indexes 13, 15 and 16 respectively. The event recognizer creates a new runtime frame (new Symbol_table and IFS) for the *print* method. The *print* method Symbol_table

has the object reference and the parameter in location 0 and 1 because the event recognizer popped the top two elements from the current method *helloWorld.main* IFS and checks the IMFS if it has any element to combine with each popped element from the IFS to handle the implicit flow. In our case the IMFS is empty. Then the event recognizer stores the popped elements in the *Print* method Symbol_table as illustrated in Figure 8.2.

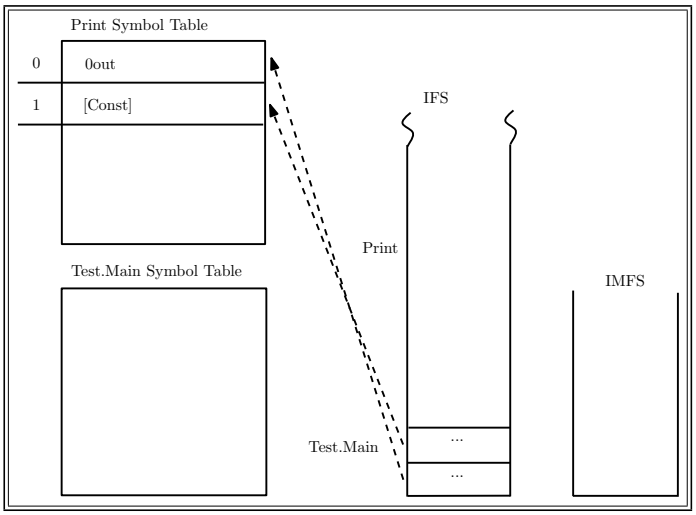


Figure 8.2: Runtime frame and IMFS of the current method Print

As mentioned in Section 5.2 only the frame of the executing method, i.e. *Print*, is active. This frame will be referred to as the current frame and its method *Print* is known as the current method. Listing 8.4 presents the original bytecode of the current method *java.io.PrintStream.print()*.

Listing 8.4: Original bytecode code of java.io.PrintStream.print

```
00: aload_1
01: ifnonnull 7
04: ldc #52 = "null"
06: astore_1
07: aload_0
08: aload_1
09: invokespecial #45 = Method java.io.PrintStream.write((Ljava/lang/String;)V)
12: return
```

Listing 8.5 shows the instrumented bytecode of method *java.io. PrintStream.print()*.

Listing 8.5: Instrumented bytecode of java.io.PrintStream.print

```
00: aload_1
01: dup
02: ldc_w #1249 = "1"
05: invokestatic #1251 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
08: invokestatic #1253 = Method Monitor.EventRecognizer.ifcond(()V)
11: ifnonnull 30
14: invokestatic #1255 = Method Monitor.EventRecognizer.Const(()V)
17: ldc #52 = "null"
19: dup
20: ldc_w #1256 = "1"
23: invokestatic #1258 = Method Monitor.EventRecognizer.Store((Ljava/lang/Object;Ljava/lang/String;)V)
26: astore_1
27: invokestatic #1270 = Method Monitor.EventRecognizer.Endif(()V)
30: aload_0
31: dup
32: ldc_w #1259 = "0"
35: invokestatic #1261 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
38: aload_1
39: dup
40: ldc_w #1262 = "1"
43: invokestatic #1264 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
46: ldc #1244 = "write"
48: iconst_1
49: iconst_0
52: invokestatic #1266 = Method Monitor.EventRecognizer.Method((II)V)
55: invokespecial #45 = Method java.io.PrintStream.write((Ljava/lang/String;)V)
58: iconst_1
59: invokestatic #1268 = Method Monitor.EventRecognizer.Return((I)V)
62: return
```

As indicated in Listing 8.5, Opcode index 05 sends an event to the event recognizer to load the contents of Symbol_table location 1 as specified in Opcode index 02. The event recognizer pushes the contents of label 1 onto the current runtime frame IFS. The second event is at Opcode index 08 that informs the event recognizer

about the *if* statement. The event recognizer pops one element from the top of IFS which is in our case the contents of label 1 [Const] and pushes it onto the top of the IMFS to control the implicit information flow of [Const] as illustrated in Figure 8.3.

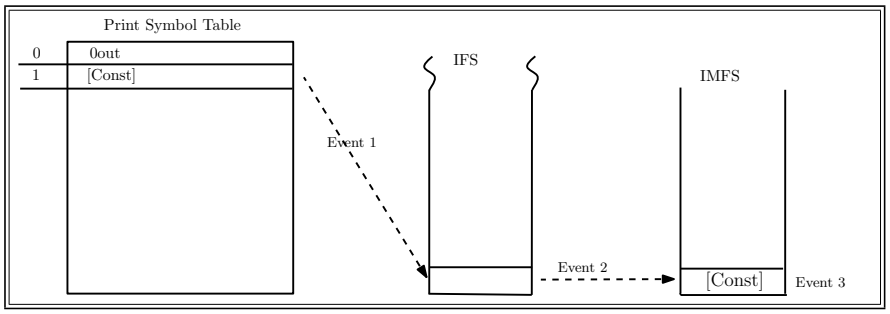


Figure 8.3: Runtime frame and IMFS of method Print

The next event will be sent according to the condition at Opcode index 11. Assuming that the condition is false, the execution flow will jump to Opcode index 27. Because our runtime monitoring creates a region for each condition statement and manipulates the instruction offset address as described in Section 5.4.
Then the third event is at Opcode index 27 that informs the event recognizer that the *if* statement ends, then the event recognizer pops the top element on the IMFS. Now both stacks (IFS and IMFS) are empty. The next event is at Opcode index 35 that sends the contents of Symbol_table location 0 as specified in Opcode index 32.

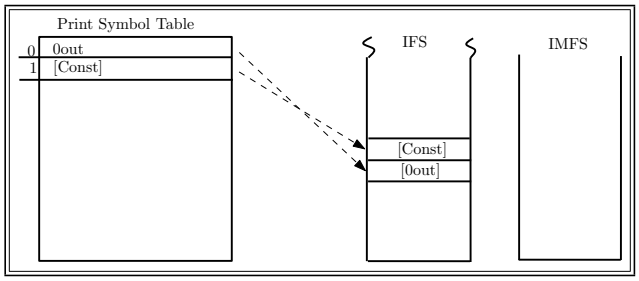


Figure 8.4: Runtime and IMFS of current method Print

The event recognizer pushes the contents of label 0 which is [0out] onto the IFS. The next call to the event recognizer is at Opcode index 43 to load the contents of Symbol_table location 1 as specified in Opcode index 40. Again the event recognizer pushes the contents of label 1 which is [Const] onto the IFS. Figure 8.4 shows the current runtime frame and IMFS of *print* method.

The next event is at Opcode index 52 that informs the event recognizer that another method is about to be invoked with the name *write*, 1 parameter and 0 return value as specified at Opcode indexes 46, 48 and 49 respectively. The event recognizer creates a new runtime frame (new Symbol_table and IFS) for the *write* method.

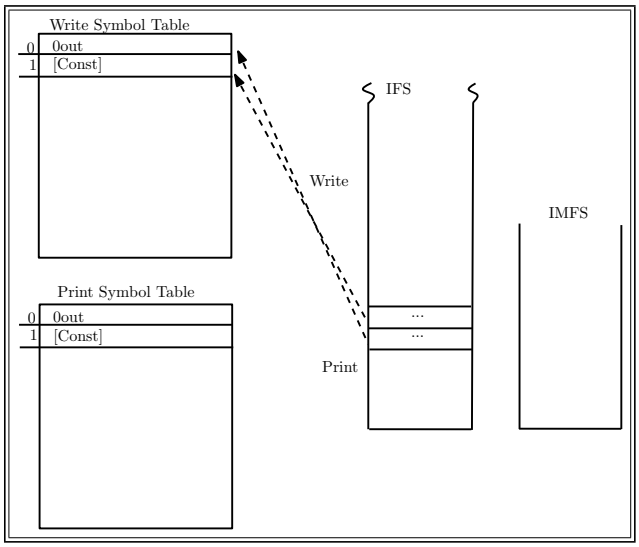


Figure 8.5: Runtime and IMFS of current method Write

The event recognizer will pop the top two elements ([Const], [0out]) from the current method IFS and checks the IMFS if there is any element to be combined with the popped elements from IFS, in our case IMFS has no element. Figure 8.5 shows the runtime frame and IMFS of the current method *write*. The execution will continue in this way creating a runtime frame for each invoked method and pass the parameters between the methods frame until returning from the method to destroy its runtime frame. Listing 8.6 presents a snapshot of the instrumented bytecode

of the method *java.io.Writer.write*, where our runtime monitoring mechanism will intercept.

Listing 8.6: A snapshot of instrumented bytecode of java.io.Writer.write

```
227: aload_0
228: dup
229: ldc_w #336 = "0"
232: invokestatic #338 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
235: aload 5
237: dup
238: ldc_w #339 = "5"
241: invokestatic #341 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
244: invokestatic #343 = Method Monitor.EventRecognizer.Const(()V)
247: iconst_0
248: iload_3
249: dup
250: ldc_w #344 = "3"
253: invokestatic #346 = Method Monitor.EventRecognizer.Iload((ILjava/lang/String;)V)
256: ldc_w #347 = "write"
259: iconst_3
260: iconst_0
261: invokestatic #349 = Method Monitor.EventRecognizer.NativeWrite((Ljava/lang/String;II)V)
264: invokevirtual #7 = Method java.io.Writer.write(([CII)V)
```

As indicated in Listing 8.6, Opcode index 232 sends an event to the event recognizer to load the contents of label 0 as specified at Opcode index 229. The event recognizer gets the contents of label 0 [0out] from the current method Symbol_table and pushes it onto the top of IFS of the current method *Write*. Opcode index 241 sends another event to load the contents of label 5 [Const] as specified at Opcode index 238. The event recognize pushes [Const] onto the top of IFS. Opcode index 244 sends an event to load constant [Const]. The event recognizer pushes [Const] onto the IFS. The next event is at Opcode index 253 to load the contents of label 3. The event recognizer gets the contents of label 3 [Const] and pushes it onto the top of the IFS as shown in Figure 8.6.

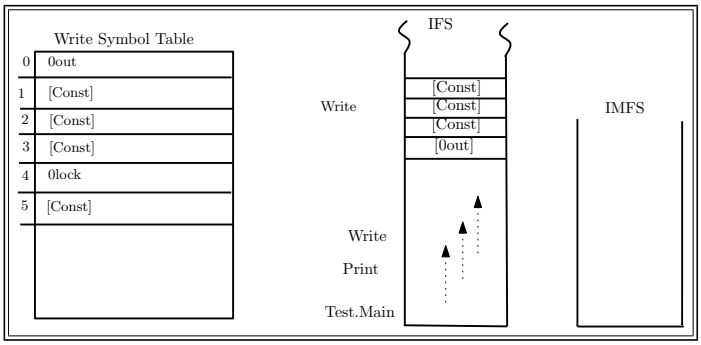


Figure 8.6: Current runtime frame and IMFS at Opcode index 253

Opcode index 261 informs the event recognizer that a **NativeWrite** method is about to be invoked with the name *java.io.Write.write*, 3 parameter and 0 return value as specified at Opcode indexes 256, 259 and 260 respectively. The event recognizer pops the top four elements from the IFS 3 parameters and the object reference ([Const], [Const], [Const] and [0out]) and sends them to the runtime checker. The runtime checker will find out that constants are going to flow to the **System.out.** In this case no need to check the information flow policy and send a message to the UserFeedBack component. Therefore, the execution will continue as normal without any intercept from our run time mechanism. Figure 8.7 shows the runtime frame and IMFS of the current method *java.io.Writer.write* when the execution is at Opcode index 261.

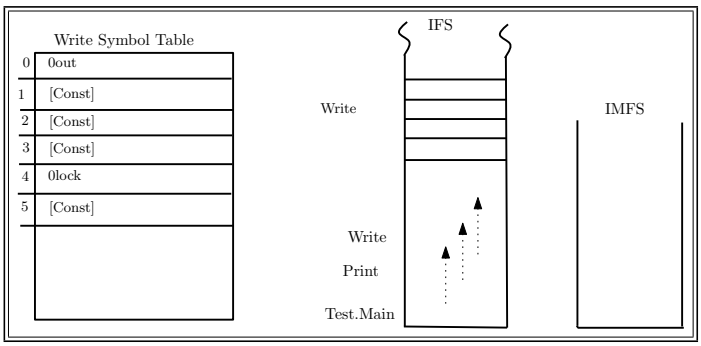


Figure 8.7: Current runtime and IMFS frame at line 261

8.3 Case Study 2

To show the feasibility of our approach, this case study presents a peer to peer file sharing application. In this case study peers are programs that can share information (files) over the network with other known peers. A peer can transfer files from the local machine to remote peers using sockets as a means of communication for the transfer itself. Each peer is an interactive program, asking the user for a file to transfer to a destination in the network. Once entered the program will open, load and transfer the file in sizeable chunks to the peer at the destination address. The case study will show how the information flow is controlled for a single peer executing on the user's behalf. The programs that are used in the scope of this case study, the original byte codes and the instrumented byte codes are presented in Appendix D. Snapshots of the instrumented bytecode will be provided in this section to show how our runtime monitoring mechanism can trace and control the flow of information. The information flow policy of this case study is as shown in Listing 8.7.

Listing 8.7: Case study 2 information flow polic

```
/* Case study 2 information flow policy – Written By Mohamed Sarrab, 27/12/10
 */
+/home/msarrab/* >>> 127.1.66.122:2000
?/home/Secret/SecretInfo.s >>> 146.227.66.150:2000
Conflict: +− >>> ?
          +? >>> ?
          −? >>> ?
           +−? >>> ?
```

Listing 8.7 shows the information flow policy for this case study, which consist of two rules the first rule states that any source from folder */home/msarrab/* is allowed to flow to destination *127.1.66.122:2000*. The second rule states that the user should be asked if the source */home/Secret/SecretInfo.s* is attempt to flow to destination *146.227.66.150:2000*. The source code of the client class is as shown in

Listing C.7.

Listing 8.8: Source code of Kclient.java

```
1  public class Kclient {
2      public static void main(String[] args) throws IOException {
3          Socket kkSocket = null;
4          BufferedReader stdIn = new BufferedReader(new InputStreamReader(
5                  System.in));
6                   while (true) {
7              System.out.println("Enter file destination <ip:port>");
8              String dest = stdIn.readLine();
9              String destsplit[] = dest.split(":");
10             if (destsplit.length != 2) {
11                 System.out.println("Wrong format! Try again.");
12                 continue;
13             }
14             try {
15                 kkSocket = new Socket(destsplit[0], Integer.parseInt(destsplit[1]));
16             }
17             catch (IOException e) {
18                 System.err.println("Couldn't get I/O for the connection to: " + dest);
19                 continue;
20             }
21                 System.out.println("Enter Source file name:");
22                 String f = stdIn.readLine();
23                 File file = new File(f);
24                 if (file.exists()) {
25                         FileInputStream fis = new FileInputStream(file);
26                         OutputStream os = kkSocket.getOutputStream();
27                         byte b = 0;
28                         while (b != 1) {
29                                 b = (byte) fis.read();
30                                 os.write(b);
31                                 }
32                         os.flush();
33                         os.close();
34                 } else {
35                         System.out.println("The specified file is not exist");
36                         System.exit(0);
37                 }
38                 kkSocket.close();
```

```
39                  }
40          }
41  }
```

The given program in Listing C.7 has only one method named *Kclient.main*. Table 8.2 shows a snapshot of the possible execution flow of class *Kclient.java*

Table 8.2: A snapshot of the execution flow of class Kclient.java

Client.main()
⟶.....
⟶.....
⟶java.net.Socket.getOutputStream()
⟶.....
⟶java.io.FileInputStream.read()
⟶**java.io.OutputStream.write()**
⟶java.io.OutputStream.flush()
⟶java.io.OutputStream.close()
⟶java.net.Socket.close()

Listing 8.9 shows the first snapshot of the instrumented bytecode of the method *Kclient.main*, when the client program asks the user to enter the IP address and port number of the machine.

Listing 8.9: First snapshot of instrumented bytecode of Kclient.main

```
66: ldc #194 = "0out"
68: invokestatic #196 = Method Monitor.EventRecognizer.LoadField((Ljava/lang/String;)V)
71: getstatic #35 = Field java.lang.System.out(Ljava/io/PrintStream;)
74: invokestatic #198 = Method Monitor.EventRecognizer.Const(()V)
77: ldc #39 = "Enter file destination <ip:port>"
79: ldc #199 = "println"
81: iconst_1
82: iconst_0
83: invokestatic #203 = Method Monitor.EventRecognizer.Method((Ljava/lang/String;II)V)
86: invokevirtual #41 = Method java.io.PrintStream.println((Ljava/lang/String;)V)
89: aload_2
90: dup
```

```
91: ldc #204 = "2"
93: invokestatic #207 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
96: ldc #208 = "readLine"
98: iconst_0
99: iconst_1
100: invokestatic #210 = Method Monitor.EventRecognizer.Method((Ljava/lang/String;II)V)
103: invokevirtual #47 = Method java.io.BufferedReader.readLine(()Ljava/lang/String;)
106: dup
107: ldc #212 = "3"
109: invokestatic #214 = Method Monitor.EventRecognizer.Store((Ljava/lang/Object;Ljava/lang/String;)V)
112: astore_3
```

Figure 8.8 illustrates the runtime frame of the current method *Kclient.main* and IMFS when the execution is at Opcode index 66 of Listing 8.9 of the method *Kclient.main*.

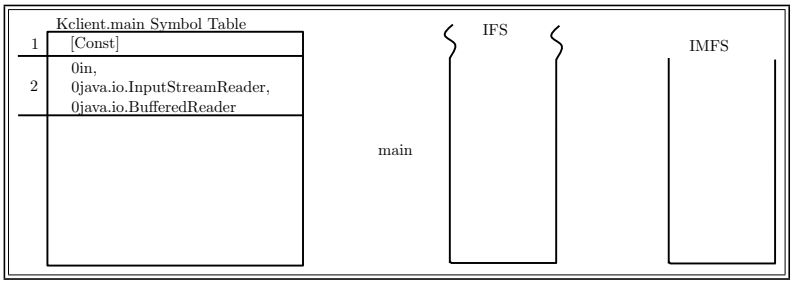


Figure 8.8: Current runtime frame and IMFS at Opcode index 66

Opcode index 68 sends an event to the event recognizer to load field named *0out* which means *System.out*. The event recognizer pushes *0out* onto the IFS of the current method. Next event is at line 74 and informs the event recognizer to push constant [Const] onto the top of the IFS. Figure 8.9 show the runtime frame and the IMFS of the current method after pushing ([0out], [Const]).

At Opcode index 83 the event recognizer is informed about the next invoked method named *println* with one parameter and no return values as specified at Opcode indexes 79, 81, 82 respectively in Listing 8.9. The event recognizer pops the top two elements (One parameter [Const] and Object reference *[0out]*) from the current method *Kclient.main* IFS and checks the contents of the IMFS to control

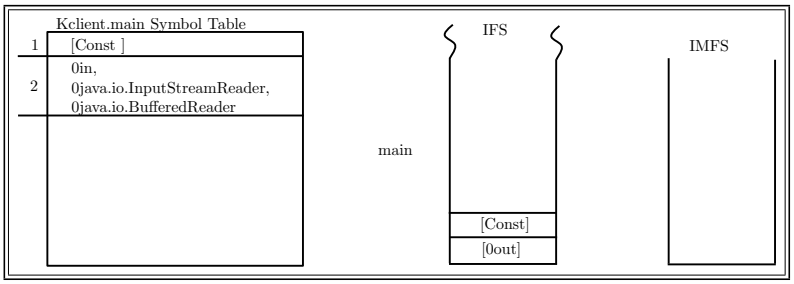


Figure 8.9: Current runtime frame and IMFS at Opcode index 77

the implicit information flow. Then it sends the popped elements to the new created Symbol_table of the invoked method *println*.

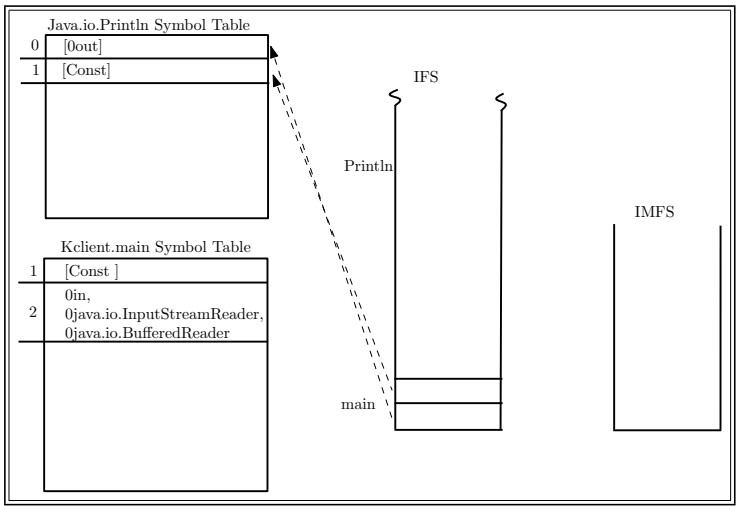


Figure 8.10: Current runtime frame and IMFS at Opcode index 86

The trace of the information flow of the *Println* method is as has been described in the first case study 8.2. Figure 8.10 shows the runtime frame and the IMFS at the invocation of *Println* Opcode index 86 of Listing 8.9.

Assuming that the execution has been returned from the *Println* method. The next event is at Opcode index 93 and informs the event recognizer to load the contents of label 2 onto the IFS of the current method *Kclient.main*. Opcode index 100 sends an event to inform the event recognizer that the next invoked method is *readline* with no parameter and one return value as indicated at Opcode indexes 96, 98 and 99 of Listing 8.9. The event recognizer pops one element (*0 parameters and the*

object reference) from the IFS and again checks the contents of the IMFS for any implicit flow and then sends the popped elements (0in, 0java.io.InputStreamReader and 0java.io.BufferedReader) to the new runtime frame Symbol table.

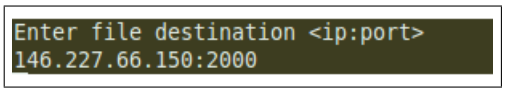


Figure 8.11: User enters file destination

Assuming that the execution returns from *readline* which returns a line of text.

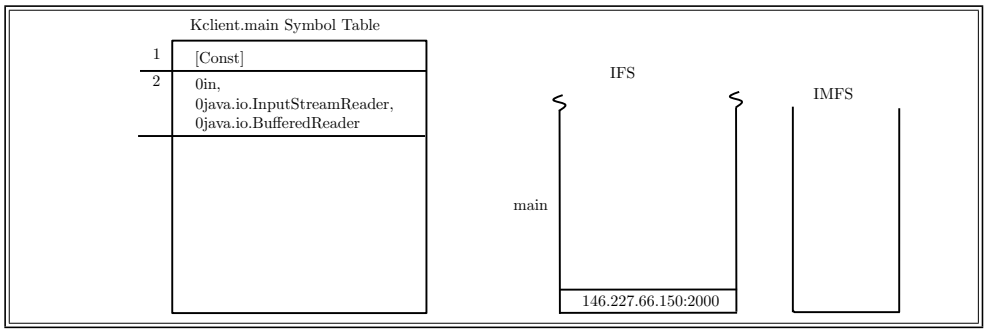


Figure 8.12: Current frame after return from method readLine

As shown in Figure 8.11, the entered line of text will be pushed onto the top of IFS of the mother method *Kclient.main* as it is the return value from the *readLine* method as specified at Opcode index 99 in Listing 8.9. Figure 8.12 shows the current runtime frame and IMFS after return from the *readLine* method. The next event is at Opcode index 109 of Listing 8.9 that informs the event recognizer to pop the top element from the IFS and store it in the Symbol_table location 3 of the current method *Kclient.main*. Figure 8.13 shows the runtime frame of the current method and IMFS when the execution is at Opcode index 112.

Suppose that the execution is at Opcode index 354 of the current method *Kclient.main*. Listing 8.10 shows a snapshot of the instrumented bytecode of the method *Kclient.main*, when the client program asks the user to enter the source file name.

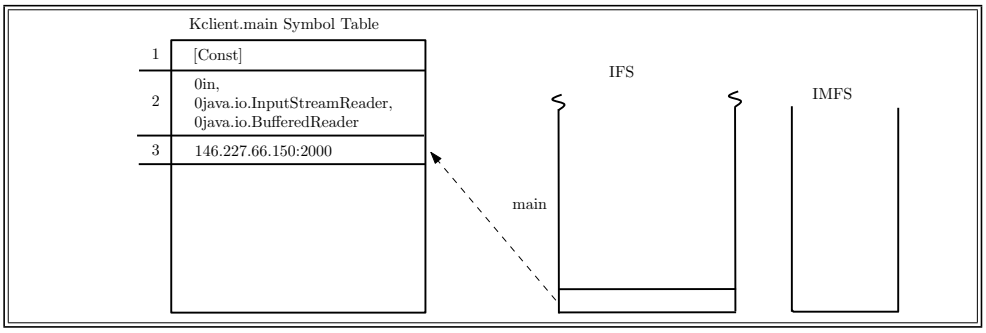


Figure 8.13: Current runtime frame and IMFS at Opcode index 112

Listing 8.10: Second snapshot of instrumented bytecode of Kclient.main

```
354: ldc_w #302 = "0out"
357: invokestatic #304 = Method Monitor.EventRecognizer.LoadField((Ljava/lang/String;)V)
360: getstatic #35 = Field java.lang.System.out(Ljava/io/PrintStream;)
363: invokestatic #306 = Method Monitor.EventRecognizer.Const(()V)
366: ldc #88 = "Enter Source file name:"
368: ldc_w #307 = "println"
371: iconst_1
372: iconst_0
373: invokestatic #309 = Method Monitor.EventRecognizer.Method((Ljava/lang/String;II)V)
376: invokevirtual #41 = Method java.io.PrintStream.println((Ljava/lang/String;)V)
379: aload_2
380: dup
381: ldc_w #310 = "2"
384: invokestatic #312 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
387: ldc_w #313 = "readLine"
390: iconst_0
391: iconst_1
392: invokestatic #315 = Method Monitor.EventRecognizer.Method((Ljava/lang/String;II)V)
395: invokevirtual #47 = Method java.io.BufferedReader.readLine(()Ljava/lang/String;)
398: dup
399: ldc_w #316 = "5"
402: invokestatic #318 = Method Monitor.EventRecognizer.Store((Ljava/lang/Object;Ljava/lang/String;)V)
405: astore 5
```

Opcode indexes 357, 363, 373, 384 and 392 send events to the event recognize is similar to the scenario of entering IP address and port number as indicated in Listing 8.10 but in this case for the source file name as shown in Figure 8.14. Assuming that the execution returns from *readline* method and is at Opcode index 395 of Listing

8.10 which returns a line of text.

```
Enter file destination <ip:port>
146.227.66.150:2000
Enter Source file name:
/home/Secret/SecretInfo.s
```

Figure 8.14: User enters file name

Figure 8.15 illustrates the changes in the current runtime frame and IMFS after returning from method *readline*.

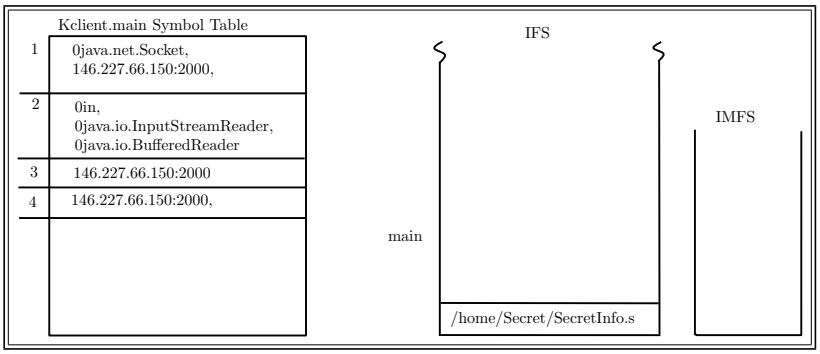


Figure 8.15: Current method runtime frame and IMFS at Opcode index 395

The next event is at Opcode index 402 that informs the event recognizer to pop the contents of IFS and combine it with the IMFS elements and then store them in the Symbol_table location 5 of the current method. Figure 8.16 shows the runtime frame of the current method and IMFS when the execution is at Opcode index 405.

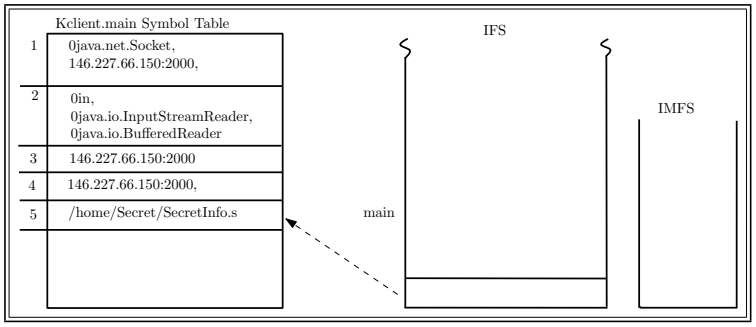


Figure 8.16: Current runtime frame and IMFS at Opcode index 405

Listing 8.11 shows a snapshot of the instrumented bytecode of *Kclient.main* method, when the runtime monitoring mechanism intercept before the file flows to the target socket.

Listing 8.11: Third snapshot of instrumented bytecode of Kclient.main

```
582: aload 8
584: dup
585: ldc_w #385 = "8"
588: invokestatic #387 = Method Monitor.EventRecognizer.Aload((Ljava/lang/Object;Ljava/lang/String;)V)
591: iload 9
593: dup
594: ldc_w #388 = "9"
597: invokestatic #391 = Method Monitor.EventRecognizer.Iload((ILjava/lang/String;)V)
600: iconst_1
601: iconst_0
602: invokestatic #394 = Method Monitor.EventRecognizer.NativeWrite((II)V)
605: invokevirtual #110 = Method java.io.OutputStream.write((I)V)
```

Figure 8.17 shows the runtime frame of the current method *Kclient.main* and IMFS when the execution is at Opcode index 582.

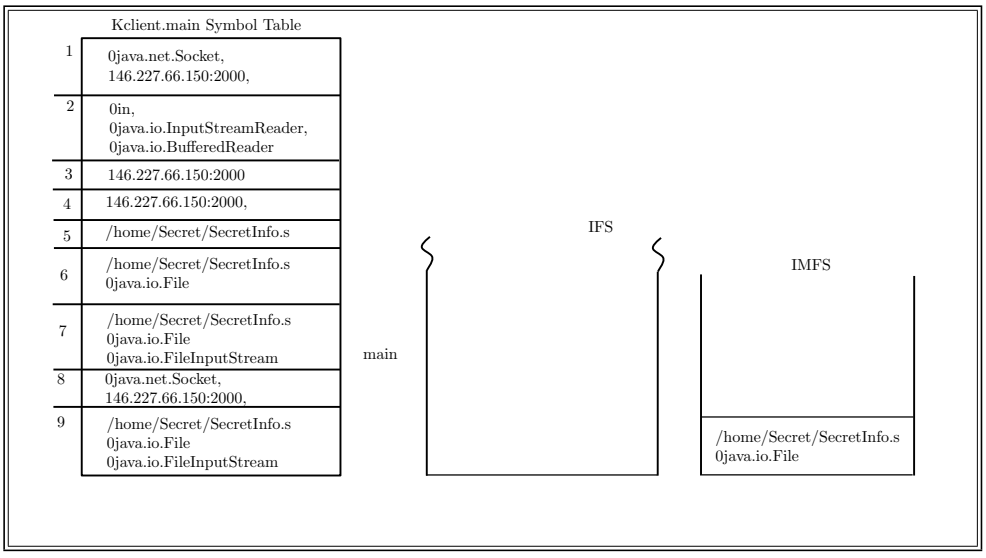


Figure 8.17: Current method runtime frame and IMFS at line 582

Opcode index 588 in Listing 8.11 sends an event to load the contents of label 8 as specified in Opcode index 585. The event recognizer gets the contents of label

8 from the current method Symbol_table (*[0java.net.Socket, 146.227.66.150:2000]*) and push it onto the IFS. Opcode index 597 sends an event to load the contents of label 9. The event recognizer gets the contents of label 9 from the Symbol_table of the current method (*[/home/Secret/SecretInfo.s, 0java.io.File, 0java.io.FileInputStream]*) and pushes them onto the top of the IFS. Figure 8.18 illustrates the changes in the current runtime frame and IMFS at Opcode index 597.

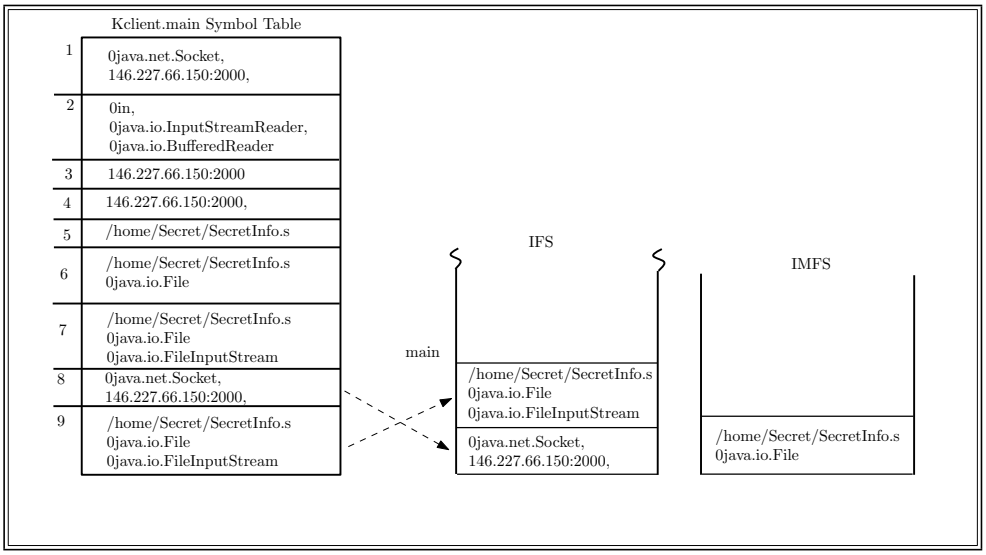


Figure 8.18: Current method runtime frame and IMFS at Opcode index 597

Opcode index 602 informs the event recognizer that the next invoked method is *NativeWrite* with 1 parameter and 0 return value. The event recognizer pops the top two elements from the IFS ([/home/Secret/SecretInfo.s, 0java.io.File, 0java.io. FileInputStream] and [0java.net.Socket, 146.227.66.150:2000]) and all elements of the IMFS ([/home/Secret/SecretInfo.s, 0java.io.File]).

Figure 8.19 shows the runtime frame of the current method and IMFS when the execution is at Opcode index 602. The event recognizer destroys any value's name started by 0 and sends the remaining contents (/home/Secret/SecretInfo.s, 146.227. 66.150:2000) to the runtime Checker. The runtime Checker checks this flow against the information flow policy in Listing 8.7. The second rule in the policy stated that:

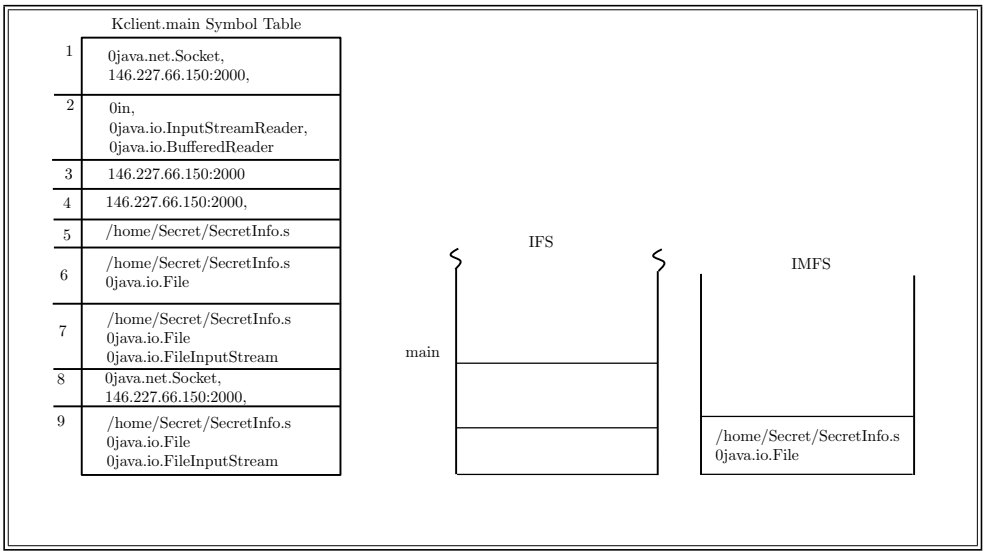


Figure 8.19: Current method runtime frame and IMFS at Opcode index 602

?/home/Secret/SecretInfo.s >>> 146.227.66.150:2000

The user should be asked for the flow of the source */home/Secret/SecretInfo.s* to destination *146.227.66.150:2000*. Then the runtime Checker will monitor the flow and ask the user as illustrated in Figure 8.20.

```
Enter file destination <ip:port>
146.227.66.150:2000
Enter Source file name:
/home/Secret/SecretInfo.s
Do you want the flow from:
/home/Secret/SecretInfo.s >>>  146.227.66.150:2000
```

Figure 8.20: Monitoring the flow

Our runtime monitoring mechanism asks the user for the next step as shown in Figure 8.20. If the user rejects the flow then the runtime monitoring mechanism stops the program execution and if the user allow the flow the runtime Checker modifies the information flow policy according to the user's decision.

8.4 Summary

The presented chapter has discussed two case studies that describe how information flow can be traced and controlled. The first case study has been provided to show how the instrumentation process, event recognizer and runtime checker components interact together to find out any possible information flow within a Java application. The general aim of this case study was to show how our runtime monitoring mechanism works and how information flow can be traced. The second case study has been provided to show the feasibility of our approach to control the information flow in a file sharing application, where the actual flows that take place at runtime are traced and the program is only interrupted when a policy violation does occur. This means that even unsafe programs may be executed within safe parameters, i.e. as long as they do not actually violate the information flow policy. The main purpose of this case study was to demonstrate the work of our approach to control the information flow.

Chapter 9

Evaluation

Objectives

- Evaluate the research which has been described in this thesis.
- Discuss the limitations of the proposed approach.

9.1 Introduction

While the previous Chapters 4, 5, 6 discussed some implementation characteristics relating to individual components, this chapter examines the overall system performance. The goal is to provide a runtime verification framework for monitoring and controlling information flow within Java applications. This chapter will evaluate our approach based on the success criteria:

- Feasibility of implementation.
- User ability to modify the flow policy during runtime in response to incidents.
- Modifying the behaviour of the program that is leaking confidential information according to the user decision.
- Performance overhead, which is broken down further into overhead in computation time and overhead in memory usage.

The feasibility of the implementation is discussed in Section 9.2. Section 9.3 evaluates the user ability to modify the flow policy during runtime in response to incidents. Section 9.4 discusses the modification of the program behaviour that is leaking confidential information according to the user decision. The examples showing the performance overhead associated with information flow control is described in Section 9.5.

9.2 Evaluating the Feasibility of Implementation

A straightforward prototype implementation without any optimisation has been described in Chapter 7. The prototype shows how the components of the proposed framework interact together to load, instrument and control the flow of the information in the target class file or class files. Limitations of the prototype are both

functional and performance related. Firstly, not all the functions and algorithms explained in Chapters 4 and 5 are implemented and thus the prototype will not be able to trace all information flows. For example the information flow introduced by exception handling mechanisms are not covered. The performance of the prototype is certainly worse than technically possible using the presented approach. For example string comparisons are not optimised and there is no code level optimisation to avoid redundant comparisons. The limitations are due to the limited amount of implementation time. This does not constitute a conceptual problem, but is due to the amount of effort that would be needed to implement the runtime verification framework for controlling information flow based on policies. This is out of scope of this PhD project.

Of course this implementation presents only a prototype to show that monitoring at runtime can be done with an acceptable performance hit (see section 9.5). It is not implemented to a level at which it could be readily commercially exploited. For example the following functionalities are currently missing or are limited in their applicability:

- Exception, *athrow* instruction deals with exceptions, but since the current prototype does not support it yet, so the proposed framework will first have to be implemented completely, before *athrow* can do anything useful.

- The Jump subroutine *jsr_w* and *jsr* are for jump to subroutines are missing. The execution proceeds at that offset from the address of the instruction. These instructions can be implemented similar to the *goto* instruction as described in Section 4.10.4 but *jsr* needs two branch addresses and *jsr_w* needs four branches addresses.

- Modifying an information flow policy needs more work since the current prototype has not a fully working version.

In order to give an insight into the current implementation performance and analysis results, the Section 9.5 presents some examples that are used to evaluate the prototype performance.

9.3 The User Ability to Modify the Flow Policy

The conceptually most important component is the user feedback component that acts as an interface between a user and the monitored application. An essential functionality of the user feedback component is that all user interaction passes through this component. The user feedback component informs the user about any feedback received from the runtime checker, if the runtime checker determined that this execution would violate the information flow policy then it sends feedback to the user through the user feedback component, the information flow policy will be modified according to the user decision. In other words, while the application is running, the user feedback component receives feedback from the runtime checker (Steering). If the application is about to enter an insecure state then the user will be asked to determine whether the information flow should be aborted or allowed to flow and continue under a modified policy. One of the motivation of this research is that with most of the previous work the information flow policy was not in the hand of the end user.

9.4 Modifying the Behaviour of the Program

Modifying the behaviour of the program that is leaking confidential information according to the user decision. An interaction with the user is very important in flexible and reliable information flow control systems because different users may have different security requirements. An interaction with monitoring mechanism

during runtime enables the user to change application behaviours or modify the way that information flow while the program is executing.

9.5 Performance

The ability to monitor and control information flow within a Java application comes at the expense of performance overhead. This overhead is introduced at two distinct points: memory usage and computation time. The experiments that measures the performance overhead were conducted on an Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz, running Ubuntu 10.04.1 LTS with the 2.6.32-27-generic (i686) Linux Kernel.

9.5.1 Memory Usage

This section discusses the target class size before and after instrumentation, memory used during class loading as part of the instrumentation code for the target classes, dynamic overhead that may be caused by the runtime classes of the proposed framework and finally it discusses the overall memory used.

In the following we present four programs that measure this overhead. The first program is a *helloworld* program and the second is another version of the *helloworld* program that prints *hello world* ten times. The third one considers a program that opens a file, reads the contents one line at a time and prints the entire contents to the screen. The file size is 1.6 KB. The last program involves measuring performance of a QuickSort algorithm for 1000 integer random numbers. The full details and the code of these examples can be found in Appendix B.

9.5.1.1 Class Size

As the bytecode of the original target program is instrumented when loading its classes, the memory size of the target program is expected to be increased when executing within the framework. In Chapter 4 the instrumentation of bytecode instructions that cause information flow has been discussed in detail. Revisiting the presented algorithms 4.9.1 to 4.10.4 it is clear that every instrumentation will add at least 74 bytes to the original bytecode in the case of a constant instrumentation, see Section 4.9.1. In the worst case 106 bytes are added to the original bytecode, in the case of Invokevirtual and Invokestatic operations, see Section 4.9.14.3. Consequently we expect a program to expand by at least a factor 1 and at most 4. The set up of the class size experiment is as follows:

- Measure target program size.
- Instrument target program (only target classes)
- Measure size of instrumented target classes.

The results of this experiment show that:

Table 9.1: Comparing original and instrumented class size

Experiment	Original size	Instrumented size	Expansion Ratio
1	582 bytes	778 bytes	1.3
2	690 bytes	1334 bytes	1.9
3	1843 bytes	3328 bytes	1.8
4	2612 bytes	6423 bytes	2.4

The results show that the instrumented bytecode of the target program increases the size. The first program has the minimum expansion ratio 1.3 due to the fact that the target program consist of only one method. That program has four bytecode instructions, one of these bytecode instructions is instrumented 1 to 4 but the others

are instrumented 1 to 1 and 1 to 2. Program 4 has a larger expansion ratio 2.4 than the other programs due to the fact that the target program involves eight methods.

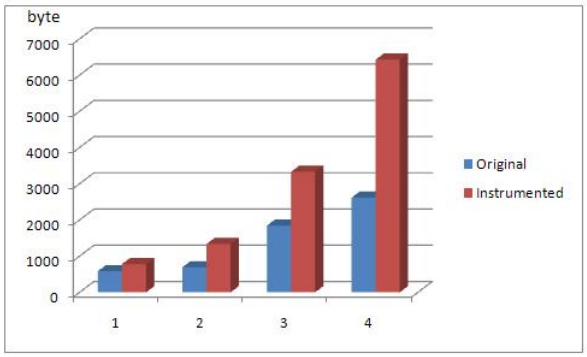


Figure 9.1: Comparison of original and instrumented class size

These methods have many operations such as (Invokevirtual and Invokestatic) which are instrumented as 1 to 4 instructions and many load and store operations that instrumented as factor 1 to 3 instructions. However, the obtained results are as expected, i.e. instrumentation will increase the target program size by at least a factor 1 and at most 4. All assertion points calls in our prototype implementation are inserted as class methods that use static (early) binding. With a better implementation that changes all assertion points to be inserted as instance methods that use dynamic (late) binding may reduce the overall size.

9.5.1.2 Memory Used during Class Loading as part of the Instrumentation Code

The set up of the experiment is as follows:

- Measure free memory size after run JVM.
- Instrument target program (only target classes).

- Measure free memory size after instrumenting target classes.

The results of this experiment are as the follows:

Table 9.2: Memory usage of original and instrumented class

Experiment	Original class	Instrumented class	Expansion Ratio
1	243880 bytes	487856 bytes	2.0
2	243899 bytes	731760 bytes	3.0
3	251898 bytes	1223504 bytes	4.8
4	487704 bytes	1955752 bytes	4.0

The results show that compared original and instrumented bytecode of the target program increases the size of the memory used during loading stage. The third program has a larger expansion ratio 4.8 than the other programs because its target class requires more classes to be loaded in order to execute. All these classes are instrumented during loading stage as explained in Section 4.5.

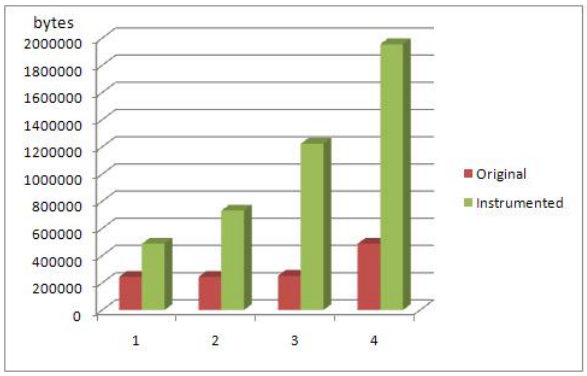


Figure 9.2: Memory usage of original and instrumented class

9.5.1.3 Dynamic Overhead

The runtime classes of the proposed framework are *Event Recognizer*, *Runtime Checker* and *MyStack*. Table 9.3 lists each of these runtime classes and their size.

Table 9.3: The size of framework runtime classes

Class	Size
Event Recognizer	8929 bytes
Runtime Checker	2512 bytes
MyStack	2465 bytes

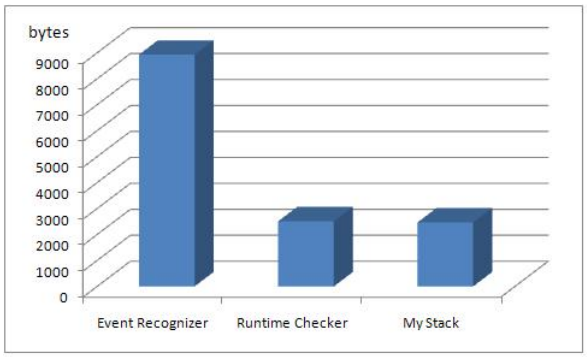


Figure 9.3: Size of framework runtime classes

As indicated in Table 9.3 the *Event recognizer* has the largest size due to the fact that all assertion points operations are defined in this class and all trace data are manipulated in this class which increases the memory usage during runtime. As explained in Chapter 5 a new runtime frame is created each time a method is invoked. The runtime frame consists of an information flow stack (IFS) and a Symbol_Table. At any point of the execution, there are thus likely to be many frames and equally many information flow stacks (IFS) that as expected will increase the used memory compared to the original application used memory. Thus, the size is as big as the target program and the data manipulated in the runtime frame.

9.5.1.4 Overall Memory usage

The set up of the experiment as follows:

- Measure free memory size.
- Instrument user and system classes (target and rt.jar classes).
- Measure free memory size after instrumenting user and system classes.

Table 9.4: Original and instrumented overall Memory usage

Experiment	Original	Instrumented	Expansion Ratio
1	2071646 bytes	11829419 bytes	5.7
2	2086133 bytes	12138858 bytes	5.8
3	2286748 bytes	12852374 bytes	5.6
4	2136934 bytes	13526455 bytes	5.6

As indicated in Table 9.4 that the lowest expansion ratio 5.6 is in the third and fourth programs and the highest is in the second program. The amount of overall used memory is increased because the measurement of the memory size is done after all framework classes, the Java agent class, transformer class, the javaassist package classes are loaded and all other loaded classes are instrumented during loading stage. The amount of overall memory usage can be reduced using static analysis techniques such as (Banerjee & Naumann 2005, Myers 1999) for the system classes. That analysis how information will flow in the program to determine whether it obeys some predefined policy with respect to an information flow without running the program.

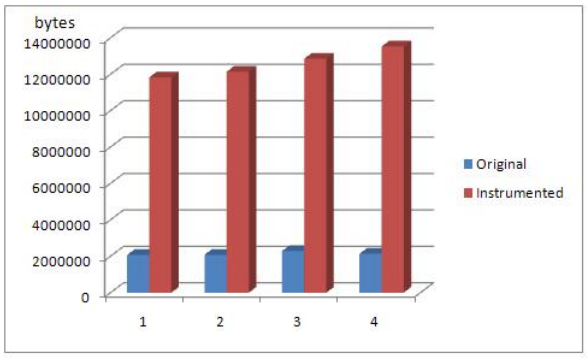


Figure 9.4: Original and instrumented overall Memory usage

Overall the experiments show that the memory usage are different between the original target program and the same program executing within the proposed framework due to two main factors. The first factor is that programs loaded different number

of required classes that they demand to run. The other factor is the Java garbage collection. The JVM's heap stores all objects created by the executing program. The garbage collection is the process that automatically free objects that are no longer referenced by the program. In addition to freeing unreferenced objects, a garbage collector may also combat heap fragmentation. Heap fragmentation occurs through the course of normal program execution. New objects are allocated, and unreferenced objects are freed such that free blocks of heap memory are left in between blocks occupied by live objects. Requests to allocate new objects may have to be filled by extending the size of the heap even though there is enough total unused space in the existing heap.

As can be seen in the Appendix B that running the same program many time may uses different amount of memory. These are the main factors impacting on the memory overhead of the proposed framework is confirmed by measuring the overall memory usage of both the original target program and the same program executing within the runtime verification framework for controlling information flow

9.5.2 Computation Time

This section discusses the overhead in loading the target classes, rt.jar classes and the overhead in executing assertion points. The same presented four programs that are used to measure the expansion ratio of the memory usage will be used to measure the computation time. The details of these programs can be found in Appendix B.

9.5.2.1 Overhead in loading and instrumenting target classes

During the loading stage of the target program classes, the framework classes, Javaagent class, transformer class and Javaassist package classes, all required system classes (rt.jar classes) will be loaded and the target program classes will be loaded and instrumented. Consequently, the expected time for loading a program is ex-

panded by at least a factor 5 and at most 11. The set up of the loading target classes experiment is as follows:

- Measure the elapsed time for loading the target classes.
- Measure the elapsed time for loading and instrument the target classes.

Table 9.5: Target classes loading and instrumented time

Experiment	Loading and Instrumenting Time	Loading time	Expansion Ratio
1	0.043 seconds	0.0055 seconds	7.8
2	0.0561 seconds	0.0059 seconds	9.5
3	0.0905 seconds	0.0089 seconds	10.1
4	0.1230 seconds	0.0159 seconds	7.7

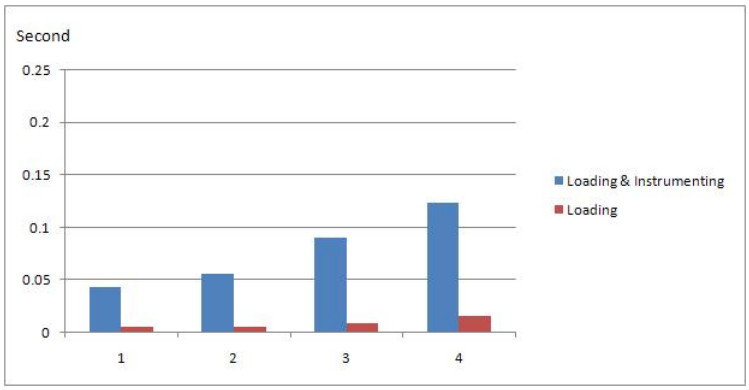


Figure 9.5: Target classes loading and instrumented time

The results show that the loading of original classes and instrumented bytecode of the target program classes increases the elapsed loading time. The third program has a larger expansion ratio (10.1) than the other programs due to the fact that loading this program requires more system classes (rt.jar classes) to be loaded than the other programs 1, 2 and 4. During the loading stage of all of these programs, the framework classes, Javaagent class, transformer class and Javaassist package classes, all required system classes (rt.jar classes) will be loaded and the target

program classes will be loaded and instrumented. Thus, that time is a reasonable time to be elapsed for loading all required class to control information flow.

9.5.2.2 Overhead in instrumenting and reloading system (rt.jar) classes

During the instrumentation stage of the system (rt.jar) classes, the framework classes, Javaagent class, transformer class and Javaassist package classes, will be loaded. The system (rt.jar) classes will be instrumented and reloaded again. However, these classes does not require any extra classes to be loaded. Thus the expected time for instrumenting and reloading system (rt.jar) classes is expand by at least a factor 1 and at most 4.

The set up of the loading target classes experiment is as follows:

- Measure the elapsed time for loading the system (rt.jar) classes.
- Measure the elapsed time for instrumenting and reloading system (rt.jar) classes.

Table 9.6: Original loading Time and Instrumenting, reloading time

Experiment	Loading and Instrumenting time	Loading time	Expansion Ratio
1	3.2496 seconds	1.0593 seconds	3.0
2	3.2960 seconds	1.0767 seconds	3.0
3	3.3055 seconds	1.1036 seconds	2.9
4	3.2893 seconds	1.1884 seconds	2.7

The results in Table 9.6 show that the loading of system (rt.jar) classes and instrumented and reloaded system (rt.jar) classes increases the elapsed loading time. The first and the second programs have the same expansion ratio 3.0 because both of them requires approximately the same number of the system (rt.jar) classes to be loaded and instrumented. Thus, the consumed time is as expected to be at least a factor 1 and at most 4. The amount of instrumentation and reloading time can be reduced using static analysis techniques such as (Banerjee & Naumann 2005, Myers 1999) for the system classes e.g. *java.lang*, *java.system* and *java.security*.

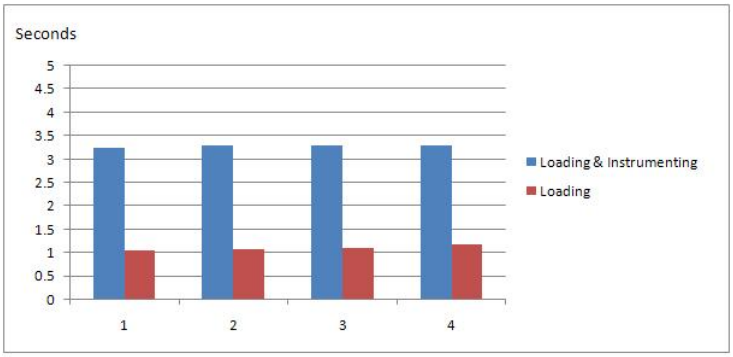


Figure 9.6: Original loading Time and Instrumenting, reloading time

9.5.2.3 Overhead in executing assertion points

The Java language provides two basic kinds of methods, instance methods and class (or static) methods. When the JVM invokes a class method, it selects the method to be invoked based on the type of the object reference, which is known at compile-time. On the other hand, when the virtual machine invokes an instance method, it selects the method to be invoked based on the actual class of the object, which can be known at run time. The JVM uses two different instructions to invoke instance or static methods:

- *invokevirtual* for instance methods, that pops the *objectref* and *args*, invoke the method at constant pool index
- *invokestatic* for class methods, that pops *args*, invoke the static method at constant pool index.

The proposed approach inserts assertion points as a static method call. Table 9.7 shows the results of a calling class method *invokestatic* and instance method *invokevirtual* doing nothing for a number of times.
The set up of the experiment is as follows:

- Measure the elapsed time for n times calling the static method.
- Measure the elapsed time for n times calling the instance method.

Table 9.7: Time cost of static and instance methods

Experiment	Number loop	Static Method	Instance Method
1	1000	0.0 seconds	0.0 seconds
2	10000	0.0 seconds	0.001 seconds
3	100000	0.002 seconds	0.004 seconds
4	1000000	0.006 seconds	0.007seconds

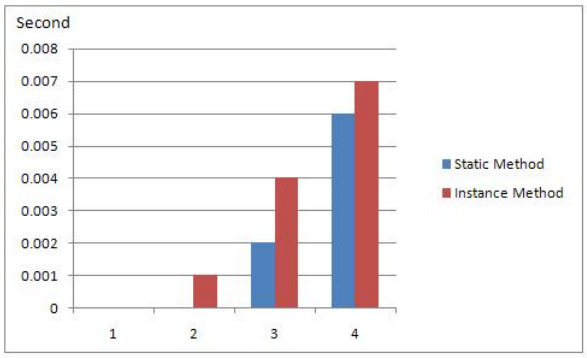


Figure 9.7: Time cost of static and instance methods

Table 9.8 compares the elapsed time for calling static and instance methods. The results show that in the first experiment equal time from both methods instance and static. However, in the other experiments the elapsed time for calling instance method is always greater than the static methods. Due to the fact the calling instance method involves overhead that the program must first examine the object to determine its type, select the appropriate method, and then call it. The elapsed time for calling any type of methods (instance or static) always depends on its number of arguments (parameters).

9.5.2.4 Overhead in manipulating runtime frames' stacks

As described in Section 5.2 the proposed approach creates a new runtime frame each time a method is invoked. The runtime frame consists of the information flow stack (IFS) and a Symbol_Table for use by the current method. At any point in the execution, there are thus likely to be many frames and equally many information flow stacks (IFS) per method invocation and one implicit information flow stack (IMFS) to control implicit information flow. Thus, the expected elapsed time for manipulating information flow stack is expanded by at least a factor 3 and at most 7. The set up of the experiment is as follows.

- Measure the execution time of the original classes.
- Measure the execution time of the original classes using our approach.

Table 9.8: Execution time of original classes and using our approach

Experiment	Original	Using our approach	Expansion Ratio
1	0.0008 seconds	0.0025 seconds	3.1
2	0.001 seconds	0.0066 seconds	6.6
3	0.0518 seconds	0.3438 seconds	6.6
4	0.2334 seconds	0.8253 seconds	3.5

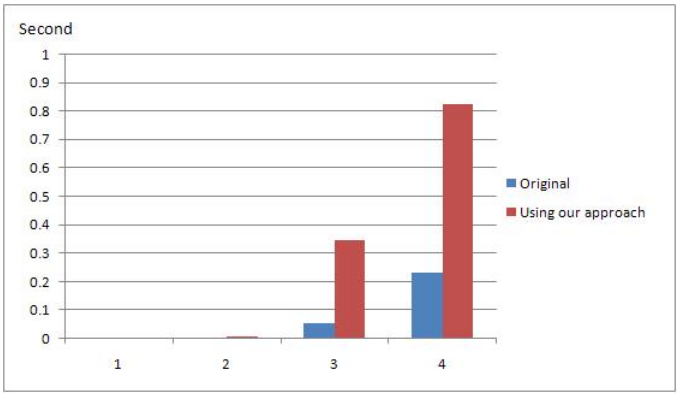


Figure 9.8: Execution time of original classes and using our approach

The results in Table 9.8 show that the first experiment has the smallest expansion ratio 3.1 and the largest expansion ratio 6.6 is for experiments 2 and 3. These expansion ratios depends on the number of operations that the inserted assertion point has, e.g. *const* assertion point has one operation that push a label to the information flow stack however, *union* assertion point pops the top two labels from the information flow stack, combines them and push them as one element to the information flow stack as described in Sections 5.3.1 and 5.3.10 respectively.
Next experiment shows a comparison between the execution time using our approach and the execution time using our approach for only explicit information flow. As explained in Section 5.2.3 the proposed approach uses a shared implicit information flow stack to control any possible implicit information flow. Thus, we expected no much time to be added for manipulating implicit information flow stack. The set up of the experiment is as follows.

- Measure the execution time of the original classes using our approach.
- Measure the execution time of the target classes using our approach for controlling explicit information flow.

Table 9.9: Execution time using our approach with/without implicit information flow

Experiment	With implicit	Without implicit	Expansion Ratio
1	0.0025 seconds	0.0023 seconds	1.0
2	0.0066 seconds	0.0058 seconds	1.1
3	0.3438 seconds	0.3410 seconds	1.0
4	0.8253 seconds	0.7976 seconds	1.0

The results in Table 9.9 show that the largest expansion ratio 1.1 is at experiment 2. Generally, as expected there is not much time added for manipulating a shared implicit information flow stack.

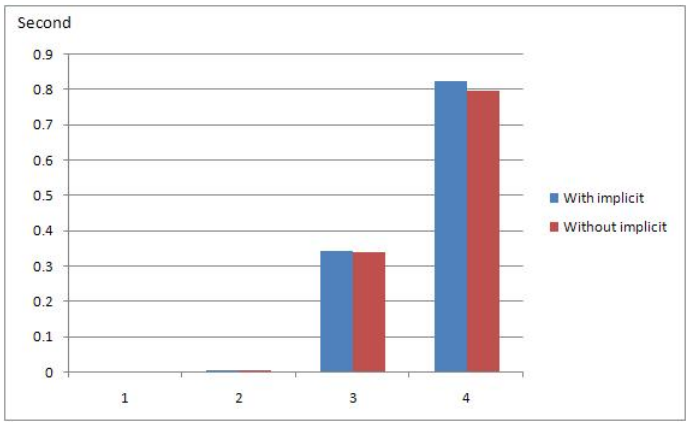


Figure 9.9: Execution time using our approach with/out implicit information flow

9.5.3 Summary of Experiments

The results of the overall experiments show that the original and the same program executing within the proposed framework introduces an overhead in load and execution time. The overhead is due to two main factors. The first factor is that programs load a different number of required classes that it originally demanded to run. The other factor is that the proposed approach instruments all loaded classes and reloads them again to be executed after inserting the required assertion points. While this is not a negligible overhead, we feel that this penalty is a reasonable price to pay for the additional security offered by the information flow control and policy enforcement functionality. We are confident that the overhead can be reduced by further optimizations in the bytecode instrumentation and runtime monitoring mechanism.

Reducing the overhead that are due to all loaded classes instrumentation during compilation time could potentially improve performance furthers, e.g. loading time can be reduced using static information flow analysis for all system classes. Furthermore, the execution time can be further reduced by skipping the repeated assertion points and assertion points that are inside a loop could potentially called once.

9.6 Summary

The core aim of this work was to develop a runtime verification framework for controlling information flow based on policies within a Java application. This chapter has analysed and evaluated the proposed framework. The beginning of this chapter considered the feasibility of the prototype implementation The second part of this chapter evaluates the performance overhead, which was broken down into overhead in memory usage and overhead in computation time. Memory usage discusses target class size before and after instrumentation, memory used during class loading as part of the code instrumentation, dynamic overhead, overall memory usage. Computation time discusses the overhead in loading and instrumenting target classes, instrumenting and reloading system (rt.jar) classes, executing assertion points and manipulating runtime frames' stacks.

Chapter 10

Conclusion and Future Work

Objectives

- Summary of the research which has been described in this thesis.
- Propose future work.

10.1 Summary of the Thesis

The thesis presented a new framework for policy-based runtime verification of information flow that supports user interaction during runtime and explicitly states how the framework components interact to control the flow of information (Chapter 3). The thesis described a flexible approach to information security management so that the information flow within a program execution conforms to a defined set of information flow rules (Chapter 6).

The thesis discussed how to control information flows during untrusted program execution. The approach concentrates on providing a dynamic and adaptable information security solution by interacting with the user during system execution in response to information flow events. This approach is advantageous over static verification as it is configurable and also places control in the hand of the user.

The thesis provided a new instrumentation algorithm to monitor and trace the program execution, that is applicable to any Java bytecode (Chapter 4). The thesis presented new algorithms for dynamically tracing and controlling the flow of the information during runtime, using a new runtime monitoring technique to control both explicit and implicit information flow (Chapter 5).

The presented approach does not treat the application as a black box (with the general assumption that once information has passed into it can find its way to any destination). Instead the actual flows that take place at runtime are traced and the program is only interrupted when a policy violation does occur. This means that even *unsafe* programs, i.e. as long as they do not actually violate the information flow policy, can be executed within safe parameters.

The presented approach works for Java bytecode and does not require any modification to the underlying hardware/software architecture. The prototype implementation of the presented approach is independent of any specific JVM. The verification in this approach is performed inside the Java environment and does not rely on any extra verification tool.

Finally, the thesis demonstrated that security can be achieved by an interactive process in which the presented framework queries the user for specific information as security requirements (Chapter 6). These are made available to the software and are then enforced on the application using a novel runtime verification technique for tracing information flow.

10.2 Achieving Success Criteria

To answer the research questions that were highlighted in Chapter 1, a new framework for policy-based runtime verification of information flow that supports user interaction has introduced:

- User ability to modify the flow policy during runtime. The thesis discussed the user interaction with the monitoring mechanism in Chapters (Chapter 3, 6 and 8). The interaction with users is very important in any flexible and reliable security monitoring mechanism because different users may have different security requirements. One of the motivations of this research is that most of the previous work, the information flow policy is not in the hand of the end user. To the best of our knowledge it is the first monitoring mechanism for controlling information flow during runtime that support user interaction.

- Modifying the behaviour of the program that is leaking confidential information according to the user decision. Our monitoring mechanism enables the user to interact with the monitoring mechanism to change the program behaviours or modify the way that information flow while the program is executing.

- The functionality and performance of the prototype tool were evaluated by employing it in several cases studies. The performance results indicates that the prototype tool is suitable for use within Java applications.

10.3 Contributions

The main contribution of this research is a novel usable security mechanism for controlling information flow within a software application during runtime. Usable security refers to enabling users to manage their systems security without defining elaborate security rules before starting the application. Security will be achieved by an interactive process to enforce user requirements on the application using runtime verification technique for tracing information flow. The contributions are detailed in the following list.

- **Runtime Monitoring**: The proposed runtime monitoring mechanism ensures that the program execution are contains only legal flows that are defined in the information flow policy or approved by the user. The presented approach provides a high degree of flexibility to support detecting and monitoring of potential leaking behaviour of a program and the user decides whether to abort or continue the program execution. The approach ensures that the program contains only those flows approved by the user.

- **Runtime Management**: The behaviour of a program that about to leak confidential information will be altered by the monitor according to the user decision. The interaction process enables users to manage their security requirements during runtime.

- **User interaction control**: The achieved user interaction with the monitoring mechanism during runtime enable users to change the program behaviours while the program is executing. The presented approach provides a high degree of flexibility to support the user's ability to modify the way that information flow.

10.4 Future Work

The most immediate need in our current prototype implementation is the dynamic policy modification by the user. Currently, the information flow policy is simply written out as code that is read by our prototype implementation. The prototype implementation should be able to manage complex information policies that can succinctly capture a wide range of user information flow requirements.

Performance measurements using the proposed approach show that the system incurs expected overhead. Optimisations are needed to reduce this overhead, some of which have been outlined in Section 9.2.

The prototype would require an expansion of the implementation to include more of Java bytecode features, e.g. exceptions are heavily used in Java, and can introduce new control flows, and therefore new indirect leaks. Other works have already addressed this issue (Myers 1999, Pottier & Simonet 2003), and it should not be too difficult to add these to the proposed approach.

Another future work direction is the examination of how much information is leaked, and the amount of the information flow to allow users to make more informed decisions.

The presented research only addresses the information flow within one application. This is sufficient to protect against untrusted applications executed in user space, but fails to address mandatory information flow control within open multi-user environments. One aspect of future work that is already addressed in ongoing research projects is the integration of the proposed approach in a wider information system context.

Bibliography

Aarniala, J. (2005), 'Instrumenting java bytecode', *Seminar work for the Compiler-scourse, Department of Computer Science, University of Helsinki, Finland* .

Agarwal, A., Sites, R. & Horowitz, M. (1986), 'Atum: A new technique for capturing address traces using microcode', *Proceedings of the 13th International Symposium on Computer Architecture* pp. 199–127.

Agat, J. (2000), 'Transforming out timing leaks', *In Proc. ACM Symp. Principles Programming Languages* pp. 40–53.

Anderson, R. (2001), *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley Computer Publishing.

Andrews, G. & Reitman, R. (1980), 'An axiomatic approach to information flow in parallel programs', *ACM TOPLAS* **2(1)**, 56–75.

Ball, T. (1993), 'What's in a region?: or computing control dependence regions in near-linear time for reducible control flow', *ACM Letters on Programming Languages and Systems (LOPLAS). New York, NY, USA* **2**, 1–16.

Bandara, A., Lupu, E. & Sloman, M. (2007), *handbook of Network and System Administration, chepter Policy Based Management*, Elsevier.

Banerjee, A. & Naumann, D. (2002), 'Secure information flow and pointer confine-

ment in a java-like language', *In Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW 15). Cape Breton, Nova Scotia* pp. 253–267.

Banerjee, A. & Naumann, D. (2005), 'History-based access control and secure information flow', *In Proceedings of the workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS)* **3362 of Lecture Notes in Computer Science, Springer-Verlag, March 2005**, 27–48.

Beres, Y. & Dalton, C. I. (2003), 'Dynamic label binding at run-time', *Proceedings of the 2003 Workshop on New security paradigms* pp. 39–46.

Bieber, P. & Cuppens, F. (1992), 'A logical view of secure dependencies', *Journal of Computer Security, 1(1):99-129, 1992.* **1(1)**, 99–129.

Binder, W., Hulaas, J. & Moret, P. (1973), 'Secure computer systems: Mathematical foundations', *MITRE Corp., Bedford, MA, MTR-2547* **1**.

Binder, W., Hulaas, J. & Moret, P. (2006), 'A quantitative evaluation of the contribution of native code to java workloads', *In 2006 IEEE International Symposium on Workload Characterization (IISWC-2006), San Jose, CA, USA* .

Binder, W., Hulaas, J. & Moret, P. (2007*a*), 'Advanced java bytecode instrumentation', *In PPPJ 2007 (5th International Conference on Principles and Practices of Programming in Java), Lisbon, Portugal, 2007. ACM Press* p. 135144.

Binder, W., Hulaas, J. & Moret, P. (2007*b*), 'Reengineering standard java runtime systems through dynamic bytecode instrumentation', *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM-2007), Paris, France, IEEE Computer Society, ISBN 0-7695-2880-5* pp. 91–100.

Birznieks, G. (1998), 'Perl taint mode'.
URL: *http://gunther.web66.com/FAQS/taintmode.html.*

Bishop, M. (2003), *Computer Security Art and Science*, Boston, Mass. ; London : Addison-Wesley.

Brewer, D. & Nash, M. (1989), 'The chinese wall security policy', *In Proceedings of the IEEE Symposium on Security and Privacy (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif* pp. 215–228.

Brown, J. & Knight, T. (2001), 'A minimal trusted computing base for dynamically ensuring secure information flow.', *Technical Rep ARIES-TM-015, MIT* .

Cavadini, S. & Cheda, D. (2008), Run-time information flow monitoring based on dynamic dependence graph, *in* 'IEEE Computer society.'.

Chander, A., Mitchell, J. & Shin, I. (2001), 'Mobile code security by java bytecode instrumentation', *In Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX'01) IEEE Computer Science Press* pp. 1027–1040.

Chandra, D. (2006), Information Flow Analysis and Enforcement in Java Bytecode, PhD thesis, University of California, Irvine.

Chiba, S. (2007), 'Class loader', *Available from http://www.csg.is.titech.ac.jp/chiba/javassist/tutorial/tutorial.html [Accessed 15/06/09]* .

Chiba, S. & Nishizawa, M. (2003), 'An easy-to-use toolkit for efficient java bytecode translators', *Proceedings of the 2nd International on Generative Programming and Component Engineering (GPCE '03)* **2830**, 364–376.

Cohen, G., Chase, J. & Kaminsky, D. (1998), 'Automatic program transformation with joie', *In Proceedings pf the 1998 USENIX Annual Technical Symposium* .

Denning, D. (1975), Secure information flow in computer systems, PhD thesis, Purdue University, CSD TR 145.

Denning, D. (1976), 'A lattice model of secure information flow', *Communications of the ACM* **19(5)**, 236–243.

Denning, D. & Denning, P. (1977), 'Certification of programs for secure information flow', *Communication, ACM* **20(7)**, 504–513.

DeveloperWorks (2001), 'Understanding the java classloader', *Free Java tutorials & programming source code http://ibm.com/developerWorks* .

Fagan, M. (1986), 'Advances in software inspections', *IEEE Transactions on Software Engineering* **12(7)**, 744–751.

Fenton, J. (1974*a*), 'An abstract computer model demonstrating directional information flow,', *University of Cambridge* .

Fenton, J. S. (1974*b*), 'Memory less subsystems', *The Computer Journal* **17(2)**, 143–147.

Ferrante, J., Ottenstein, K. & Warren, J. (1987), 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems* **9(3)**, 319–349.

Genaim, S. & Spoto, F. (2005), 'Information flow analysis for java bytecode', *International Conference on Verification, Model Checking and Abstract Interpretation* pp. 346–362.

Goguen, J. & Meseguer, J. (1982), 'Security policies and security models', *In Proceedings of the Symposium on Security and Privacy, IEEE Computer Society Press* pp. 11–20.

Guernic, G. (2009), 'Precise dynamic verification of confidentiality', *INRIA-MSR - Parc Orsay Universit, 91893 Orsay - France. http://www.msr-inria.inria.fr/gleguern* .

Haldar, V., Chandra, D. & Franz, M. (2005), 'Practical, dynamic information flow for virtual machines', *2nd International Workshop on Programming Language Interference and Dependence* .

Hastings, R. & Joyce, B. (1992), 'Purify: Fast detection of memory leaks and access errors', *Proceedings of the Winter USENIX Conference* pp. 125–136.

Havelund, K. & Goldberg, A. (2005), 'Verify your runs', *Verified Software: Theories, Tools, Experiments, (VSTTE05), Zurich, Switzerland* pp. 10–13.

Herrmann, P. (2001), 'Information flow analysis of component structured applications.', *IEEE Computer Society Press* pp. 45–54.

Hoffman, L. & Davis, R. (1990), 'Security pipeline interface (spi)', *In Proc. 6th Annual Computer Security Applications Conference (ACSAC 1990), Tuscon, AZ, USA* **.**, 349–355.

Horwitz, S., Reps, T. & Binkley, D. (1990), 'Interprocedural slicing using dependence graphs', *ACM Transactions on Programming Languages and Systems* **12(1)**, 35–46.

H.Saltzer, J. & D.Schroeder, M. (1975), 'The protection of information in computer systems', *IEEE* **63(9)**, 1278–1308.

Janicke, H. (2007), The Development of Secure Multi-Agent Systems, PhD thesis, Software Technology Research Laboratory, Department of Computer Science and Engineering, De Montfort University, Leicester.

Janicke, H., Siewe, F., Jones, K., Cau, A. & Zedan, H. (2005), Analysis and run-time verification of dynamic security policies, *in* 'Proceedings of the 1st Workshop on Defence Applications for Multi-Agent Systems (DAMAS2005) Utrecht, Netherlands'.

Joshi, A. (2009), Java bytecode instrumentation for security monitoring, Master's thesis, De Montfort University, Leicester, UK.

Joshi, R. & Leino, K. R. (2000), 'A semantic approach to secure information flow', *Secience of Computer Programming* **37(1-3)**, 113–138.

Keller, R. & Horlzle, U. (1998), 'Binary componeny adaption', *In Proceeding of the 1998ECOOP* .

Kim, M., Viswanathan, M., BenAbdallah, H., Kannan, S., Lee, I. & Sokolsky, O. (1999), Mac: A framework for run-time correctness assurance of real-time systems., *in* 'Philadelphia, PA, Department of computer and Information Science University of Pennsylvania.'.

Korel, B. & Laski, J. (1988), 'Dynamic program slicing', *Information Processing Letters* **29(3)**, 155–163.

Lam, L. & Chiueh, T. (2006), 'A general dynamic information flow tracking framework for security applications', *Proceedings of the 22nd Annual Computer Security Applications Conference, Washington, DC, USA* pp. 463–472.

LaPadula, L. J. & Bell, D. E. (1973), 'Secure computer systems: A mathematical model', *MITRE Corp., Bedford, MA, MTR-2547* **2**, Reprinted in J. Comput. Security, vol. 4, no. 23, pp. 239263, 1996.

Larus, J. & Ball, T. (1994), 'Rewriting executable files to measure program behavior.', *Software, Practice and Experience* **24(2)**, 197–218.

Lee, H. & Zorn, B. (1997), 'Bit: A tool for instrumenting java bytecodes', *Proceedings of the USENIX Symposium on Internet Technologies and Systems Monterey, California.* .

Lee, I., BenAbdallah, H., Kannan, S., Kim, M. & Viswanathan, M. (1998), A monitoring and checking framework for run-time correctness assurance.

Leino, K. & Joshi, R. (1998), 'A semantic approach to secure infor mation flow', *In Proc. Mathematics of Program Construction* **1422 of LNCS**, 254–271.

Liang, S. (1999), *The JavaTM Native Interface Programmers Guide and Specification*, ADDISON-WESLEY.

Liang, S. & Bracha, G. (1998), 'Dynamic class loading in the java virtual machine', *In Proceedings of Object-Oriented Programming, Systems, Languages & Applications* **33- no 10 of ACM**, 36–44.

Lindholm, T. & Yellin, F. (1997), *The JavaTM Virtual Machine Specification*, Sun Microsystems.

Lipton, R. & Snyder, L. (1977), 'A linear time algorithm for deciding subject security', *Journal of the ACM* **24(3)**, 455464.

Masriand, W. & Podgurski, A. (2005), 'Using dynamic information flow analysis to detect attacks against applications', *In Proceedings of the 2005 Workshop on Software Engineering for secure sytems Building Trustworthy Applications, ACM, New York, USA* .

McCullough, D. (1988), 'Noninterference and the composability of security properties', *In Proceedings of IEEE Symposium. Security Privacy* pp. 177–186.

Meyer, J. & Dowing, T. (1997), 'Java virtual machine.', *O'Reilly and Associates, Inc. Sebastopol, CA, USA* .

Michael Fagan Associates (2010).

Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, B., Karavanic, K., Kunchithapadam, K. & Newhall, T. (1995), 'The paradyn parallel performance measurement tool', *IEEE Computer, Special issue on performance evaluation tools for parallel and distributed computer systems* **28(11)**, 37–46.

Myers, A. (1999), 'Jflow: Practical mostly-static information flow control', *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99), San Antonio, Texas* p. 228241.

Nagatou, N. & Watanabe, T. (2006), 'Run-time detection of covert channels', *IEEE Computer Society* pp. 577–584.

NASA (1993), *Software Formal Inspections Guidebook, NASA-GB-A302, Office of safety and mission assurance*, NASA-GB-A302.

Newsome, J. & Song, D. (2005), 'Taint analysis for automatic detection, analysis and signature generation of exploits on commodity software', *12th Network and Distributed System Security Symposium* .

Oracle (2010), 'Java platform standard ed. 6, package java.lang.instrument', *Java SE Developer Documentation.* .

P.Allen (1991), 'A comparison of non-interference and non-deducibility using csp', *In Proceedings of the Fourth IEEE Computer Security Foundations Workshop, Franconia, New Hampshire,* pp. 43–54.

Parthasarathy, S., Cierniak, M. & Li., W. (1996), 'Netprof: Network-based high-level profiling of java bytecode.', *Technical Report 622, Computer Science Department, University of Rochester* .

Pottier, F. & Conchon, S. (2000), 'Information flow inference for free', *In Proc. ACM Int. Conf. Functional Programming* pp. 46–57.

Pottier, F. & Simonet, V. (2003), 'Information flow inference for ml.', *ACM Trans. on Programming Languages and Systems* **25(1)**, 117–158.

Sabelfeld, A. & Myers, A. (2003), 'Language-based information-flow security', *IEEE Journal on Selected Areas in Communication* **21(1)**, 5–19.

Sabelfeld, A. & Sands, D. (1999), 'A per model of secure information flow in sequential programs', *In Proc. European Symposium on Programming, Springer-Verlag* **1576 of LNCS**, 40–58.

Sabelfeld, A. & Sands, D. (2000), 'Probabilistic noninterference for multi-threaded programs', *In Proc. IEEE Computer Security Foundations Workshop* pp. 200–214.

Sabelfeld, A. & Sands, D. (2001), 'A per model of secure information flow in sequential programs, higher order symbolic comput', *Higher Order Symbolic Comput* **14(1)**, 5991.

SATC (2002), 'Satc: Software formal inspections'.
URL: *http://satc.gsfc.nasa.gov/fi/fipage.html*

Schwartz, R., Phoenix, T. & brian d foy (2005), *Learning Perl*, O'Reilly & Associates.

Shroff, P., Smith, S. & Thober, M. (2007), 'Dynamic dependency monitoring to secure information flow', *Proceedings of the 20th IEEE Computer Security Foundations Symposium* pp. 203–217.

Simonet, V. (2003), 'Flow caml in a nutshell', *In Graham Hutton, editor, Proceedings of the first APPSEM-II workshop* pp. 152–165.

Smith, G. & Volpano, D. (1998), 'Secure information flow in a multi-threaded imperative language.', *Proceedings of the 25th ACM SIGPLAN-SIGACT* pp. 355–364.

Smith, M. (1991), 'Tracing with pixie. memo from center for integrated systems', *Stanford university* .

Srivastava, A. & Eustace, A. (1994), 'Atom: A system for building customized program analysis tools', *In Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation (PLDI)* pp. 196–205.

Srivastava, A. & Wall, D. (1993), 'A practical system for intermodal code optimization at link-time', *Journal of Programming Languages* **1(1)**, 1016.

Srivastava, A. & Wall, D. (1994*a*), 'Link-time optimization of address calculation on a 64-bit architecture', *In Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation (PLDI)* pp. 49–60.

Srivastava, A. & Wall, D. (1994*b*), 'Link-time optimization of address calculation on a 64-bit architecture.', *In Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation (PLDI)* pp. 49–60.

Sun Microsystems, I. (2000), *Trusted SolarisTM 8 Operating Environment, A Technical Overview*, http://www.sun.com/solaris.

Susan L. Graham, P. B. K. & McKusick., M. K. (1993), 'An execution profiler for modular programs.', *Software Practice and Experience* **13(1)**, 671–685.

Sutherland, D. (1986), 'Model of information', *In Proceedings of the 9th National Computer Security Conference, National Bureau of Standards and National Computer Security Center* pp. 175–183.

Tatsubori, M., Sasaki, T., Chiba, S. & Itano, K. (2001), 'A bytecode translator for distributed execution of "legacy"', *Java software. Lecture Notes in Computer Science* **2070**, 236–255.

Tilevich, E. & Smaragdakis, Y. (2002), 'J-orchestra: Automatic java application partitioning.', *In European Conference on Object-Oriented Programming (ECOOP), Malaga, Spain, June* .

Vachharajan, N., Bridges, M., Chang, J., Rangan, R., Ottoni, G., Blome, J., Reis, G., Vachharajani, M. & August, D. (2004), 'An architectural framework for user-centric information-flow security.', *Proceedings of the 37th International Symposium on Microarchitecture (MICRO), December.* .

Venners, B. (1999), *Inside the Java virtual machine*, New York; London: McGraw-Hill.

Volpano, D., Irvine, C. & Smith, G. (1996), 'A sound type system for secure flow analysis', *Computer Security* **4(2-3)**, 167–187.

Wall, D. (1992), 'Systems for late code modification tools techniques', *Springer-Verlag* pp. 275–293.

Weiser, M. (1979), Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method, PhD thesis, The University of Michigan, Ann Arbor , Michigan.

Weiser, M. (1981), 'Program slicing', *In Proceedings of the 5th International Conference on Software Engineering, San Diego, CA, IEEE Computer Society Press, New York* pp. 439–449.

Weissman, C. (1969), 'Security controls in the adept-50 timesharing system', *Proc.*

AFIPS Fall Joint Computer Conference, AFIPS Press, Montvale, N.J **35**, 119–133.

Wilson, E. (1991), *An Introduction to Scientific Research*, Dover Publications.

Wittbold, J. & M.Johnson, D. (1990), 'Information flow in nondeterministic systems.', *In Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Oakland, CA* pp. 144 – 161.

Zhang, C. & Yang, C. (2002), 'Information flow analysis on role-based access control model', *Information Management & Computer Security* **10**, 225–236.

Appendix A

Instructions and its Opcodes

Opcode	Instruction	Opcode	Instruction	Opcode	Instruction	Opcode	Instruction
0x00	nop	0x01	aconst_null	0x02	iconst_m1	0x03	iconst_0
0x04	iconst_1	0x05	iconst_2	0x06	iconst_3	0x07	iconst_4
0x08	iconst_5	0x09	lconst_0	0x0a	lconst_1	0x0b	fconst_0
0x0c	fconst_1	0x0d	fconst_2	0x0e	dconst_0	0x0f	dconst_1
0x10	bipush	0x11	sipush	0x12	ldc	0x13	ldc_w
0x14	ldc2_w	0x15	iload	0x16	lload	0x17	fload
0x18	dload	0x19	aload	0x1a	iload_0	0x1b	iload_1
0x1c	iload_2	0x1d	iload_3	0x1e	lload_0	0x1f	lload_1
0x20	lload_2	0x21	lload_3	0x22	fload_0	0x23	fload_1
0x24	fload_2	0x25	fload_3	0x26	dload_0	0x27	dload_1
0x28	dload_2	0x29	dload_3	0x2a	aload_0	0x2b	aload1_1
0x2c	aload_2	0x2d	aload_3	0x2e	iaload	0x2f	laload
0x30	faload	0x31	daload	0x32	aaload	0x33	baload
0x34	caload	0x35	saload	0x36	istore	0x37	lstore
0x38	fstore	0x39	dstore	0x3a	astore	0x3b	istore_0
0x3c	istore_1	0x3d	istore_2	0x3e	istore_3	0x3f	lstore_0
0x40	lstore_1	0x41	lstore_2	0x42	lstore_3	0x43	fstore_0
0x44	fstore_1	0x45	fstore_2	0x46	fstore_3	0x47	dstore_0
0x48	dstore_1	0x49	dstore_2	0x4a	dstore_3	0x4b	astore_0
0x4c	astore_1	0x4d	astore_2	0x4e	astore_3	0x4f	iastore
0x50	lastore	0x51	fastore	0x52	dastore	0x53	aastore
0x54	bastore	0x55	castore	0x56	sastore	0x57	pop
0x58	pop2	0x59	dup	0x5a	dup_x1	0x5b	dup_x2
0x5c	dup2	0x5d	dup2_x1	0x5e	dup2_x2	0x5f	swap
0x60	iadd	0x61	ladd	0x62	fadd	0x63	dadd
0x64	isub	0x65	lsub	0x66	fsub	0x67	dsub
0x68	imul	0x69	lmul	0x6a	fmul	0x6b	dmul
0x6c	idiv	0x6d	ldiv	0x6e	fdiv	0x6f	ddiv
0x70	irem	0x71	lrem	0x72	frem	0x73	drem
0x74	ineg	0x75	lneg	0x76	fneg	0x77	dneg
0x78	ishl	0x79	lshl	0x7a	ishr	0x7b	lshr
0x7c	iushr	0x7d	lushr	0x7e	iand	0x7f	land
0x80	ior	0x81	lor	0x82	ixor	0x83	lxor
0x84	iinc	0x85	i2l	0x86	i2f	0x87	i2d
0x88	l2i	0x89	l2f	0x8a	l2d	0x8b	f2i
0x8c	f2l	0x8d	f2d	0x8e	d2i	0x8f	d2l
0x90	d2f	0x91	i2b	0x92	i2c	0x93	i2s
0x94	lcmp	0x95	fcmpl	0x96	fcmpg	0x97	dcmpl
0x98	dcmpg	0x99	ifeq	0x9a	ifne	0x9b	iflt
0x9c	ifge	0x9d	ifgt	0x9e	ifle	0x9f	if_icmpeq
0xa0	if_icmpne	0xa1	if_icmplt	0xa2	if_icmpge	0xa3	if_icmpgt
0xa4	if_icmple	0xa5	if_acmpeq	0xa6	if_acmpne	0xa7	goto
0xa8	jsr	0xae	freturn	0xaa	tableswitch	0xa9	ret
0xac	ireturn	0xad	lreturn	0xab	lookupswitch	0xaf	dreturn
0xb0	areturn	0xb2	getstatic	0xb6	invokevirtual	0xb1	return
0xb4	getfield	0xb3	putstatic	0xb7	invokespecial	0xb5	putfield
0xb8	invokestatic	0xbd	anewarray	0xb9	invokeinterface	0xbb	new
0xbc	newarray	0xbe	arraylength	0xba	xxxunusedxxx1	0xbf	athrow
0xc0	checkcast	0xc3	monitorexit	0xc2	monitorenter	0xc1	instanceof
0xc4	wide	0xc6	ifnull	0xc5	multianewarray	0xc7	ifnonnull
0xc8	goto_w	0xc9	jsr_w	0xca	breakpoint	0xfe	impdep1
0xff	impdep2						

Appendix B

Experiments Result

Example 1:

This example considers a *HelloWorld* program.

Listing B.1: Source code of HelloWorld.java

```
package uk.ac.dmu.msarrab.vif.testtargets;
/**
 *
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class HelloWorld {
        public static void main(String[] args) {
                        System.out.println("Hello World");
        }
}
```

For the over all memory usage we achieved the following result:

Table B.1: Comparing Memory Usage

Original	Instrumented
2050512	11831128
2046720	11828928
2047920	11826968
2046696	11828928
2047920	11831128
2046696	11828928
2284408	11831583
2047920	11837064
2050896	11828856
2046776	11820680
Mean 2071646 Bytes	Mean 11829419 Bytes
Max deviation 237712 Bytes	Max deviation 16384 Bytes

For the speed we achieved the following result for ten different run:

Table B.2: Original system and user classes elapsed time

System Classes - Loading time	User Class - Loading time	Execution Time
1.067	0.005	0.001
1.061	0.005	0.000
1.048	0.005	0.001
1.066	0.005	0.001
1.063	0.004	0.001
1.046	0.005	0.001
1.059	0.007	0.000
1.057	0.006	0.001
1.070	0.007	0.001
1.056	0.006	0.001
Mean 1.0593 seconds	Mean 0.0055 seconds	Mean 0.0008 seconds
Max deviation 0.024 seconds	Max deviation 0.003 seconds	Max deviation 0.001 seconds

Table B.3: Instrumented system and user classes elapsed time

System Classes - Load and instrument time	User Class - Load and instrument time	Execution Time
3.244		0.003
3.242		0.002
3.247		0.002
3.246		0.003
3.239	0.043	0.002
3.249		0.003
3.269		0.003
3.245		0.002
3.249		0.003
3.266		0.003
Mean 3.2496 seconds	Mean 0.043 seconds	Mean 0.0025 seconds
Max deviation 0.03 seconds	Max deviation 0.0 seconds	Max deviation 0.001 seconds

Table B.4: Using our approach for only explicit information flow

Execution Time
0.002
0.002
0.002
0.003
0.002
0.002
0.003
0.002
0.003
0.002
Mean 0.0023 seconds
Max deviation 0.001 seconds

Example 2:

This example considers a program that involves one class has one method with ten lines. This example has been presented as one method with ten lines of *println* instead of using for loop due to measure the memory usage and instrumentation of the required assertion points.

Listing B.2: Source code of HelloWorldProgram.java

```
package uk.ac.dmu.msarrab.vif.testtargets;
/**
 *
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class HelloWorldProgram {
        public static void main(String[] args) {
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
                        System.out.println("Hello World");
        }
}
```

For speed we achieved the following result.

Table B.5: Comparing Memory Usage

Original	Using our approach
2040532	12327992
2264387	12067752
2041736	12070037
2036715	12330264
2040512	12249488
2284347	12078240
2036715	12067792
2041755	12061856
2037920	12065872
2036715	12069295
Mean 2086133 Bytes	Mean 12138858 Bytes
Max deviation 247632 Bytes	Max deviation 268408 Bytes

Table B.6: Original system and user classes elapsed time

System Classes - Loading time	User Class - Loading time	Execution Time
1.073	0.005	
1.076	0.005	
1.084	0.005	
1.070	0.004	
1.075	0.005	0.001
1.084	0.006	
1.081	0.005	
1.071	0.006	
1.070	0.007	
1.083	0.007	
Mean 1.0767 seconds	Mean 0.0055 seconds	Mean 0.001 seconds
Max deviation 0.014 seconds	Max deviation 0.003 seconds	Max deviation 0.0 seconds

Table B.7: Instrumented System and user classes elapsed time

System Classes - Load and instrument time	User Class - Load and instrument time	Execution Time
3.296	0.056	0.007
3.295	0.057	0.006
3.296	0.056	0.005
3.296	0.056	0.007
3.297	0.056	0.008
3.297	0.056	0.006
3.296	0.056	0.005
3.296	0.056	0.008
3.295	0.056	0.009
3.296	0.056	0.005
Mean 3.296 seconds	Mean 0.0561 seconds	Mean 0.0066 seconds
Max deviation 0.0 seconds	Max deviation 0.001 seconds	Max deviation 0.004 seconds

Table B.8: Using our approach for only explicit information flow

Execution Time
0.005
0.006
0.005
0.006
0.006
0.007
0.005
0.006
0.007
0.005
Mean 0.0058 seconds
Max deviation 0.002 seconds

Example 3:

This example considers a program that opens a file, reads the contents one line at a time and prints the entire contents into the screen. The file size is 1.6 KB.

Listing B.3: Source code of ReadFileAndPrint.java

```
package uk.ac.dmu.msarrab.target;
/**
 *
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
import java.io.*;
public class ReadFileAndPrint {
        public static void main(String[] args) throws IOException {
                try {
                        String file = "/home/msarrab/a/a1.txt";
                        File f = new File(file);
                        if (!f.exists()) {
                                System.out.println("The specified file is not exist");
                                System.exit(0);
                        } else {
                                FileInputStream finp = new FileInputStream(f);
                                byte b = 0;
                                do {
                                        b = (byte) finp.read();
                                        System.out.print(b);
                                } while (b != -1);
                                finp.close();
```

```
                }
            } catch (IOException e) {
                System.out.println(e.getMessage() + " in the specified directory.");
            }
        }
}
```

For the memory usage we achieved the following result:

Table B.9: Comparing Memory Usage

Original	Using our approach
2284336	13046232
2286784	12820108
2286784	13042008
2286784	12808392
2287928	12848387
2287928	12764576
2286744	12766776
2286744	12808280
2286704	12808392
2286744	12810592
Mean 2286748 Bytes	Mean 12852374 Bytes
Max deviation 3592 Bytes	Max deviation 281656 Bytes

For speed we achieved the following result.

Table B.10: Original system and user classes elapsed time

System Classes - Loading time	User Class - Loading time	Execution Time
1.104	0.009	0.054
1.105	0.008	0.045
1.103	0.009	0.054
1.104	0.013	0.054
1.103	0.008	0.052
1.103	0.008	0.051
1.103	0.008	0.048
1.103	0.009	0.051
1.106	0.008	0.057
1.102	0.009	0.052
Mean 1.1036 seconds	Mean 0.0089 seconds	Mean 0.0518 seconds
Max deviation 0.004 seconds	Max deviation 0.005 seconds	Max deviation 0.012 seconds

Table B.11: Instrumented system and user classes elapsed time

System Classes - Load and instrument time	User Class - Load and instrument time	Execution Time
3.3	0.087	0.341
3.305	0.096	0.345
3.297	0.088	0.346
3.301	0.089	0.342
3.320	0.089	0.345
3.290	0.089	0.341
3.325	0.088	0.345
3.308	0.095	0.347
3.315	0.095	0.342
3.297	0.09	0.344
Mean 3.3058 seconds	Mean 0.0905 seconds	Mean 0.3438 seconds
Max deviation 0.035 seconds	Max deviation 0.009 seconds	Max deviation 0.006 seconds

Table B.12: Using our approach for only explicit information flow

Execution Time
0.340
0.341
0.341
0.340
0.343
0.341
0.341
0.343
0.340
0.340
Mean 0.341 seconds
Max deviation 0.003 seconds

Example 4:

This example considers a program that involves measuring performance with Quick-Sort algorithm for 1000 integer random numbers.

Listing B.4: Source code of QuickSort.java

```
package uk.ac.dmu.msarrab.target;
/**
 *
 * @author Mohamed Sarrab (STRL, DMU, UK) Msarrab@dmu.ac.uk
 */
public class QuickSort {

        private long[] data;
        private int len;

        public QuickSort(int max) {
                data = new long[max];
                len = 0;
        }

        public void insert(long value) {
                data[len] = value;
                len++;
        }

        public void display() {
                System.out.print("Data: ");
```

```
        for (int j = 0; j < len; j++)
                System.out.print(data[j] + " ");
        System.out.println("");
    }

    public void quickSort() {
        recQuickSort(0, len - 1);
    }

    public void recQuickSort(int left, int right) {
        if (right - left <= 0) // if size <= 1 already sorted
                return;
        else // size is 2 or larger
        {
                long pivot = data[right]; // rightmost item
                // partition range
                int partition = partitionData(left, right, pivot);
                recQuickSort(left, partition - 1); // sort left side
                recQuickSort(partition + 1, right); // sort right side
        }
    }

    public int partitionData(int left, int right, long pivot) {
        int leftPtr = left - 1; // left (after ++)
        int rightPtr = right; // right-1 (after --)
        while (true) { // find bigger item
                while (data[++leftPtr] < pivot);
                // find smaller item
                while (rightPtr > 0 && data[--rightPtr] > pivot);
                if (leftPtr >= rightPtr) // if pointers cross, partition done
                        break;
                else
                        swap(leftPtr, rightPtr);
        }
        swap(leftPtr, right); // restore pivot and return pivot location
        return leftPtr;
    }

    public void swap(int d1, int d2) {
        long temp = data[d1];
        data[d1] = data[d2];
```

```
            data[d2] = temp;
        }

        public static void main(String[] args) {
            int maxSize = 1000; // array size
            QuickSort arr = new QuickSort(maxSize); // create array
            for (int j = 0; j < maxSize; j++) // fill array with random numbers
            {
                long n = (int) (java.lang.Math.random() * 99);
                arr.insert(n);
            }
            arr.display();
            arr.quickSort();
            arr.display();
        }
}
```

For the memory usage we achieved the following result:

Table B.13: Comparing Memory Usage

Original	Using our approach
2136712	13518541
2136712	13539021
2136691	13504627
2137828	13518541
2136868	13514953
2136728	13515408
2136728	13523079
2137634	13524177
2136728	13577191
2136712	13529017
Mean 2136934 Bytes	Mean 13526455 Bytes
Max deviation 1137 Bytes	Max deviation 72564 Bytes

For speed we achieved the following result.

Table B.14: Original system and user classes elapsed time

System Classes - Loading time	User Class - Loading time	Execution Time
1.186	0.018	0.240
1.186	0.015	0.208
1.191	0.018	0.207
1.188	0.014	0.217
1.190	0.015	0.243
1.188	0.018	0.221
1.189	0.017	0.253
1.191	0.015	0.245
1.185	0.015	0.221
1.190	0.014	0.280
Mean 1.1884 seconds	Mean 0.0159 seconds	Mean 0.2334 seconds
Max deviation 0.006 seconds	Max deviation 0.004 seconds	Max deviation 0.073 seconds

Table B.15: Instrumented system and user classes elapsed time

System Classes - Load and instrument time	User Class - Load and instrument time	Execution Time
3.322	0.122	0.797
3.268	0.123	0.861
3.291	0.127	0.802
3.284	0.120	0.763
3.294	0.125	0.812
3.282	0.121	0.851
3.285	0.126	0.819
3.292	0.115	0.833
3.293	0.127	0.864
3.282	0.125	0.852
Mean 3.2893 seconds	Mean 0.1230 seconds	Mean 0.8253 seconds
Max deviation 0.054 seconds	Max deviation 0.012 seconds	Max deviation 0.089 seconds

Table B.16: Using our approach for only explicit information flow

Execution Time
0.754
0.802
0.783
0.788
0.809
0.822
0.817
0.791
0.812
0.798
Mean 0.7976 seconds
Max deviation 0.068 seconds

Appendix C

Source Code

As can be seen from the source code that most of the methods and classes are declared with the access modifier public, where classes are visible to all classes everywhere and methods can be called by any object. Declaring a public method defines its access level. Of course this implementation presents only a prototype to show that monitoring at runtime can be done with an acceptable performance hit. It is not implemented to a level at which it could be readily commercially exploited. For example the the public modifier can be change to private modifier.

Listing C.1: Source code of JavaAgent.java

```
package Agent;
import java.lang.instrument.ClassDefinition;
import java.lang.instrument.Instrumentation;
import java.lang.instrument.UnmodifiableClassException;
import uk.ac.dmu.msarrab.vif.framework.EventRecognizer;
import javassist.NotFoundException;
/**
 *
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class JavaAgent {
        /**
         * JavaAgent defines already loaded classes.
```

```
 * and registers the transformer
 */
private static MyTransformer transformer = new MyTransformer();
@SuppressWarnings("unchecked")
public static void premain(String agentArgs, Instrumentation inst) {
        if (inst.isRedefineClassesSupported()) {
                {
                        System.err.println("IsRedefineClassesSupported? Supported ");
                }
                if (inst.isRetransformClassesSupported()) {
                                System.err.println("IsRetransformClassesSupported? Supported");
                                EventRecognizer.RuntimeFrame.openframe();
                        }else


                                System.err.println("IsRetransformClassesSupported? Not supported");
                Class[] loaded = inst.getAllLoadedClasses();
                for (Class<?> cc : loaded) {
                        try {
                                redefineClass(inst, cc);
                                } catch (NotFoundException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        } catch (ClassNotFoundException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        } catch (UnmodifiableClassException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
                }
        } else {
                System.err.println("isRedefineClassesSupported Not Supported");
        }
                inst.addTransformer(transformer);
        EventRecognizer.initialising = false;


}
@SuppressWarnings("unchecked")
private static void redefineClass(Instrumentation inst, Class cc)
                throws NotFoundException, ClassNotFoundException,
                UnmodifiableClassException {
```

```
                if ((cc.getName().equalsIgnoreCase("java.io.PrintStream")) ||
                  (cc.getName().equalsIgnoreCase("java.io.InputStream"))||
                 (cc.getName().equalsIgnoreCase("java.io.FileInputStream"))||
                   (cc.getName().equalsIgnoreCase("java.nio.channels.WritableByteChannel")))
                {
                    byte bytes[] = uk.ac.dmu.msarrab.vif.framework.InstrumentClass.instrument(cc.getName());
                        if (bytes != null) {
                                ClassDefinition definition = new ClassDefinition(cc, bytes);
                                 inst.redefineClasses(definition);
                                            } else {
                                            System.err.println("Class: " + cc.getName()
                                                                    + " could not be instrumented.");
                        }
                }
        }
}
```

Listing C.2: Source code of MyTransformer.java

```
package Agent;

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
/**
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
import uk.ac.dmu.msarrab.vif.framework.InstrumentClass;
import javassist.NotFoundException;

public class MyTransformer implements ClassFileTransformer {
        /**
         * all future class definitions will be seen by mytransformer.
        */
        public MyTransformer() {
                super();
        }
        public byte[] transform(ClassLoader loader, String className,
                        Class<?> redefiningClass, ProtectionDomain domain, byte[] bytes)
                        throws IllegalClassFormatException {
```

```
                char CA[] = className.toCharArray();
                System.out.println();
                System.out.println("In The Transformer " + className);
                for (int i = 0; i < CA.length - 1; i++) {
                        if (CA[i] == '/')
                                CA[i] = '.';
                }
                String classn = new String(CA);
                System.out.println("Transformer to Transform Class: " + classn);
                try {
                        if (classn.startsWith("uk.ac.dmu.msarrab.target")) // after the
                        {
                                bytes = InstrumentClass.instrument(classn);
                                System.out.println("Transfer class= " + className);

                        }
                } catch (NotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                        System.out.println("Error: " + e.getMessage());
                }
                return bytes;
        }
}
```

Listing C.3: Source code of InstrumentClass.java

```
package uk.ac.dmu.msarrab.vif.framework;
import java.io.IOException;
import javassist.CannotCompileException;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;
import javassist.Modifier;
import javassist.NotFoundException;
import javassist.bytecode.InstructionPrinter;
/**
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class InstrumentClass {
```

```
public final static boolean DEBUG = true;
/**
 * Returns the instrumented bytecode for the class named as <code>arg</code>
 * Linked to the EventRecognizer er.
 *
 * @param arg The fully classified name of the class to be instrumented
 * @param er The reference to the EventRecognizer that the instrumentation code
 *   will invoke.
 * @return The instrumented bytecode or <code>null</code> if the class could not be instrumented.
 *
 * @throws NotFoundException If class named <code>arg</code> could not be found.
 */
public static byte[] instrument(String arg)
                throws NotFoundException {
        ClassPool pool = ClassPool.getDefault();
        CtClass cc = pool.get(arg);
        if (cc.isFrozen()) {
                System.out.println("Class is frozen "+ cc);
                return null;
        } else {
                if (cc.getName().startsWith("uk.ac.dmu.msarrab.target")){
                                }
          CtMethod[] declaredMethods = cc.getDeclaredMethods();
    System.out.println("cc= "+cc);
   for (CtMethod cm : declaredMethods) {
         if(cm.getMethodInfo().isMethod()){
                                int modf = cm.getModifiers();
                                if (
                                        (!Modifier.isNative(modf))
                                        && (!Modifier.isAbstract(modf))
                                        )
                                {
                                if (DEBUG) {

                                        System.out.println();
                                        System.out.println("Original Method code: "+ cm.
                                             getMethodInfo());
                                        InstructionPrinter.print(cm, System.out);
                                         System.out.println();
                                }
                                        if (DEBUG) {
```

```
                                                System.out.println("Instrumented Method: "+cm.
                                                    getMethodInfo());
                                                }

                                            InstrumentMethod.methodInstrument(cc, cm);
                                        }
                                    }
                                }
                            }
                            byte[] bytes = null;
                            try {
                                System.out.println("getting bytecode...");
                                bytes = cc.toBytecode();
                            } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                            } catch (CannotCompileException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                            } finally {
                                cc.defrost();
                            }
                            return bytes;
                        }
}
```

Listing C.4: Source code of InstrumentMethod.java

```
package uk.ac.dmu.msarrab.vif.framework;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import javassist.CannotCompileException;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;
import javassist.Modifier;
import javassist.NotFoundException;
import javassist.bytecode.*;
import javassist.expr.ExprEditor;
import javassist.expr.MethodCall;
/**
```

```
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class InstrumentMethod {
        public final static boolean DEBUG = true;
        static int PrametersNumber = 0;
        static int ReturnNumber = 0;
        static String MethodName = null;
        static String MethodrefType = null;
        static String MethodN = null;
        static int temp = 0;
        static int Native=0;
        /**
         * Returns the instrumented bytecode for the CtMethod named as
         * <code>cm</code> Linked to the EventRecognizer er.
         *
         * @param cc
         * The CtClass of the class to be instrumented
         * @param cm
         * The CtMethod of the method to be instrumented
         * @param er
         * The reference to the EventRecognizer that the instrumentation
         * code will invoke.
         */
        public static void methodInstrument(CtClass cc, CtMethod cm) {
                        int IF_Condition = 0;
                StringBuffer sb = new StringBuffer();
                ClassPool pool = ClassPool.getDefault();
                CodeAttribute ca = cm.getMethodInfo().getCodeAttribute();
                ByteArrayOutputStream out = new ByteArrayOutputStream();
                PrintStream p = new PrintStream(out);
                InstructionPrinter.print(cm, p);
                sb.append(out.toString());
                ConstPool cp = ca.getConstPool();
                ClassFile cf = cc.getClassFile();
                try {
                        CtClass cc1 = pool.get(EventRecognizer.class.getName());
                        ClassFile cf1 = cc1.getClassFile();
                        int index = 0;
                        CodeIterator ci = ca.iterator();
                        while (ci.hasNext()) {
```

```
index = ci.next();
int op = ci.byteAt(index);
switch (op) {
case Opcode.ACONST_NULL:
case Opcode.ICONST_M1:
case Opcode.ICONST_0:
case Opcode.ICONST_1:
case Opcode.ICONST_2:
case Opcode.ICONST_3:
case Opcode.ICONST_4:
case Opcode.ICONST_5:
case Opcode.LCONST_0:
case Opcode.LCONST_1:
case Opcode.FCONST_0:
case Opcode.FCONST_1:
case Opcode.FCONST_2:
case Opcode.DCONST_0:
case Opcode.DCONST_1:
case Opcode.BIPUSH:
case Opcode.SIPUSH:
case Opcode.LDC:
case Opcode.LDC_W:
case Opcode.LDC2_W:
                MethodInfo minf1 = cf1.getMethod("Const");
        Bytecode b1 = new Bytecode(ca.getConstPool());
        b1.addInvokestatic(cc1, minf1.getName(), minf1
                        .getDescriptor());
        byte[] code = b1.get();
        ci.insert(index,code);
        break;
case Opcode.ILOAD:
        String instrI = InstructionPrinter.instructionString(ci,index, cp);
        String OpStackI3 = instrI.substring(6);
        MethodInfo mi3 = cf1.getMethod("Iload");
        Bytecode I3 = new Bytecode(ca.getConstPool());
        I3.add(89);
        I3.addLdc(OpStackI3);
        I3.addInvokestatic(cc1, mi3.getName(), mi3.getDescriptor());
        byte[] Icode3 = I3.get();
        ci.insertEx(Icode3);
        break;
```

```
                case Opcode.LLOAD:
                    String instrL = InstructionPrinter.instructionString(ci,
                            index, cp);
                    String CharOpStackL3 = instrL.substring(6);
                    MethodInfo minL3 = cf1.getMethod("Lload");
                    Bytecode L3 = new Bytecode(ca.getConstPool());
                    L3.add(89);
                    L3.addLdc(CharOpStackL3);
                    L3.addInvokestatic(cc1, minL3.getName(), minL3
                            .getDescriptor());
                    byte[] Lcode3 = L3.get();
                    ci.insertEx(Lcode3);
                    break;
                case Opcode.FLOAD:
                    String instrF = InstructionPrinter.instructionString(ci,
                            index, cp);
                    String CharOpStackF3 = instrF.substring(6);
                    MethodInfo minF3 = cf1.getMethod("Fload");
                    Bytecode F3 = new Bytecode(ca.getConstPool());
                    F3.add(89);
                    F3.addLdc(CharOpStackF3);
                    F3.addInvokestatic(cc1, minF3.getName(), minF3
                            .getDescriptor());
                    byte[] Fcode3 = F3.get();
                    ci.insertEx(Fcode3);
                    break;
            case Opcode.DLOAD:
                    String instrD = InstructionPrinter.instructionString(ci,
                            index, cp);
                    String CharOpStD3 = instrD.substring(6);
                    MethodInfo minD3 = cf1.getMethod("Dload");
                    Bytecode D3 = new Bytecode(ca.getConstPool());
                    D3.add(89);
                    D3.addLdc(CharOpStD3);
                    D3.addInvokestatic(cc1, minD3.getName(), minD3
                            .getDescriptor());
                    byte[] Dcode3 = D3.get();
                    ci.insertEx(Dcode3);
                    break;
            case Opcode.ALOAD:
                    String instrType3 = InstructionPrinter.instructionString(
```

```
                    ci, index, cp);
            char CharOpStack3 = instrType3.charAt(6);
            MethodInfo minf3 = cf1.getMethod("Aload");
            Bytecode b3 = new Bytecode(ca.getConstPool());
            b3.add(89);
            b3.addLdc(Character.toString(CharOpStack3));
            b3.addInvokestatic(cc1, minf3.getName(), minf3
            .getDescriptor());
            byte[] code3 = b3.get();
            ci.insertEx(code3);
            break;
        case Opcode.ILOAD_0:
        case Opcode.ILOAD_1:
        case Opcode.ILOAD_2:
        case Opcode.ILOAD_3:
            String inI4 = InstructionPrinter.instructionString(ci,
                    index, cp);
            String ChOpStackI4 = inI4.substring(6);
            MethodInfo mifI4 = cf1.getMethod("Iload");
            Bytecode Ib4 = new Bytecode(ca.getConstPool());
            Ib4.addLdc(ChOpStackI4);
            Ib4.addInvokestatic(cc1, mifI4.getName(), mifI4
                    .getDescriptor());
            byte[] Icode4 = Ib4.get();
            ci.insertEx(Icode4);
            break;
        case Opcode.FLOAD_0:
        case Opcode.FLOAD_1:
        case Opcode.FLOAD_2:
        case Opcode.FLOAD_3:
            String inf4 = InstructionPrinter.instructionString(ci,
                    index, cp);
            String ChOpStackf4 = inf4.substring(6);
            MethodInfo miff4 = cf1.getMethod("Fload");
            Bytecode Fb4 = new Bytecode(ca.getConstPool());
            Fb4.add(89);
            Fb4.addLdc(ChOpStackf4);
            Fb4.addInvokestatic(cc1, miff4.getName(), miff4
                    .getDescriptor());
            byte[] Fcode4 = Fb4.get();
            ci.insertEx(Fcode4);
```

```
                break;
            case Opcode.DLOAD_0:
            case Opcode.DLOAD_1:
            case Opcode.DLOAD_2:
            case Opcode.DLOAD_3:
                String ind4 = InstructionPrinter.instructionString(ci,
                        index, cp);
                String ChOpStackd4 = ind4.substring(6);
                MethodInfo mid4 = cf1.getMethod("Dload");
                Bytecode db4 = new Bytecode(ca.getConstPool());
                db4.add(89);
                db4.addLdc(ChOpStackd4);
                db4.addInvokestatic(cc1, mid4.getName(), mid4
                        .getDescriptor());
                byte[] dcode4 = db4.get();
                ci.insertEx(dcode4);
                break;
            case Opcode.ALOAD_0:
            case Opcode.ALOAD_1:
            case Opcode.ALOAD_2:
            case Opcode.ALOAD_3:
                String instrType4 = InstructionPrinter.instructionString(
                        ci, index, cp);
                String CharOpStack4 = instrType4.substring(6);
                MethodInfo minf4 = cf1.getMethod("Aload");
                Bytecode b4 = new Bytecode(ca.getConstPool());
                b4.add(89);
                    b4.addLdc(CharOpStack4);
                b4.addInvokestatic(cc1, minf4.getName(), minf4
                        .getDescriptor());
                byte[] code4 = b4.get();
                ci.insertEx(code4);
                break;
            case Opcode.LLOAD_0:
            case Opcode.LLOAD_1:
            case Opcode.LLOAD_2:
            case Opcode.LLOAD_3:
                String inT4 = InstructionPrinter.instructionString(ci,
                        index, cp);
                String ChOpStack4 = inT4.substring(6);
                MethodInfo mif4 = cf1.getMethod("Lload");
```

```
                Bytecode Lb4 = new Bytecode(ca.getConstPool());
                Lb4.add(89);
                Lb4.addLdc(ChOpStack4);
                Lb4.addInvokestatic(cc1, mif4.getName(), mif4
                        .getDescriptor());
                byte[] Lcode4 = Lb4.get();
                ci.insertEx(Lcode4);
                break;
            case Opcode.IALOAD:
            case Opcode.LALOAD:
            case Opcode.FALOAD:
            case Opcode.DALOAD:
            case Opcode.AALOAD:
            case Opcode.BALOAD:
            case Opcode.CALOAD:
            case Opcode.SALOAD:
            case Opcode.IADD:
            case Opcode.LADD:
            case Opcode.FADD:
            case Opcode.DADD:
            case Opcode.ISUB:
            case Opcode.LSUB:
            case Opcode.FSUB:
            case Opcode.DSUB:
            case Opcode.IMUL:
            case Opcode.LMUL:
            case Opcode.FMUL:
            case Opcode.DMUL:
            case Opcode.IDIV:
            case Opcode.LDIV:
            case Opcode.FDIV:
            case Opcode.DDIV:
            case Opcode.IREM:
            case Opcode.LREM:
            case Opcode.FREM:
            case Opcode.DREM:
            case Opcode.ISHL:
            case Opcode.LSHL:
            case Opcode.ISHR:
            case Opcode.LSHR:
            case Opcode.IUSHR:
```

```
case Opcode.LUSHR:
case Opcode.IAND:
case Opcode.LAND:
case Opcode.IOR:
case Opcode.LOR:
case Opcode.IXOR:
case Opcode.LXOR:
        MethodInfo minf5 = cf1.getMethod("Union");
        Bytecode b5 = new Bytecode(ca.getConstPool());
        b5.addInvokestatic(cc1, minf5.getName(), minf5
                        .getDescriptor());
        byte[] code5 = b5.get();
        ci.insert(index, code5);
        break;
case Opcode.LSTORE:
case Opcode.FSTORE:
case Opcode.DSTORE:
case Opcode.ASTORE:
case Opcode.ISTORE_0:
case Opcode.ISTORE_1:
case Opcode.ISTORE_2:
case Opcode.ISTORE_3:
case Opcode.LSTORE_0:
case Opcode.LSTORE_1:
case Opcode.LSTORE_2:
case Opcode.LSTORE_3:
case Opcode.FSTORE_0:
case Opcode.FSTORE_1:
case Opcode.FSTORE_2:
case Opcode.FSTORE_3:
case Opcode.DSTORE_0:
case Opcode.DSTORE_1:
case Opcode.DSTORE_2:
case Opcode.DSTORE_3:
case Opcode.ASTORE_0:
case Opcode.ASTORE_1:
case Opcode.ASTORE_2:
case Opcode.ASTORE_3:
        String instrType6 = InstructionPrinter.instructionString(
                        ci, index, cp);
        String CharOpStack6 = instrType6.substring(7);
```

```
                MethodInfo minf6 = cf1.getMethod("Store");
                Bytecode b6 = new Bytecode(ca.getConstPool());
                b6.add(89);
                b6.addLdc(CharOpStack6);
                b6.addInvokestatic(cc1, minf6.getName(), minf6
                                .getDescriptor());
                byte[] code6 = b6.get();
                ci.insert(index, code6);
                break;
            case Opcode.ISTORE:
                String instrType7 = InstructionPrinter.instructionString(
                                ci, index, cp);
                String CharOpStack7 = instrType7.substring(7);
                MethodInfo minf7 = cf1.getMethod("iStore");
                Bytecode b7 = new Bytecode(ca.getConstPool());
                b7.add(89);
                b7.addLdc(CharOpStack7);
                b7.addInvokestatic(cc1, minf7.getName(), minf7
                                .getDescriptor());
                byte[] code7 = b7.get();
                ci.insert(index, code7);
                break;
            case Opcode.IASTORE:
            case Opcode.LASTORE:
            case Opcode.FASTORE:
            case Opcode.DASTORE:
            case Opcode.AASTORE:
            case Opcode.BASTORE:
            case Opcode.CASTORE:
            case Opcode.SASTORE:
                MethodInfo minf8 = cf1.getMethod("astore");
                Bytecode b8 = new Bytecode(ca.getConstPool());
                b8.addInvokestatic(cc1, minf8.getName(), minf8
                                .getDescriptor());
                byte[] code8 = b8.get();
                ci.insert(index, code8);
                break;
            case Opcode.GETSTATIC:
                byte bb1 = ca.getCode()[index + 1];
                byte bb2 = ca.getCode()[index + 2];
                int index1 = (bb1 << 8) | bb2;
```

```
                String FieldName = "0"
                        + cf.getConstPool().getFieldrefName(index1);
                Bytecode be1 = new Bytecode(ca.getConstPool());
                MethodInfo minff1 = cf1.getMethod("LoadField");
                be1.addLdc(FieldName);
                be1.addInvokestatic(cc1, minff1.getName(), minff1
                        .getDescriptor());
                byte[] cd = be1.get();
                ci.insert(index,cd);
                break;
            case Opcode.GETFIELD:
                byte bee1 = ca.getCode()[index + 1];
                byte bee2 = ca.getCode()[index + 2];
                int indeex1 = (bee1 << 8) | bee2;
                String FieldN = "0"
                        + cf.getConstPool().getFieldrefName(indeex1);
                Bytecode bbe1 = new Bytecode(ca.getConstPool());
                MethodInfo minfff1 = cf1.getMethod("LoadField");
                bbe1.addLdc(FieldN);
                bbe1.addInvokestatic(cc1, minfff1.getName(), minfff1.getDescriptor()
                    );
                byte[] cdd = bbe1.get();
                ci.insert(index, cdd);
                break;
            case Opcode.PUTSTATIC:
            case Opcode.PUTFIELD:
                byte bb11 = ca.getCode()[index + 1];
                byte bb21 = ca.getCode()[index + 2];
                int index10 = (bb11 << 8) | bb21;
                String FieldName10 = "0"
                        + cf.getConstPool().getFieldrefName(index10);
                MethodInfo minf10 = cf1.getMethod("StoreField");
                Bytecode b10 = new Bytecode(ca.getConstPool());
                b10.addLdc(FieldName10);
                b10.addInvokestatic(cc1, minf10.getName(), minf10
                        .getDescriptor());
                byte[] code10 = b10.get();
                ci.insert(index, code10);
                break;


            case Opcode.NEW:
```

```
                byte bb3 = ca.getCode()[index + 1];
                byte bb4 = ca.getCode()[index + 2];
                int index33 = (bb3 << 8) | bb4;
                String ObjectRefName11 = cf.getConstPool().getClassInfo(
                                index33);
                MethodInfo minf11 = cf1.getMethod("New");
                Bytecode b11 = new Bytecode(ca.getConstPool());
        b11.addLdc(ObjectRefName11);
                b11.addInvokestatic(cc1, minf11.getName(), minf11
                                .getDescriptor());
                byte[] code11 = b11.get();
                ci.insert(index, code11);
                break;
        case Opcode.DUP:
        case Opcode.DUP_X1:
        case Opcode.DUP_X2:
        case Opcode.DUP2:
        case Opcode.DUP2_X1:
        case Opcode.DUP2_X2:
                MethodInfo minf12 = cf1.getMethod("Dup");
                Bytecode b12 = new Bytecode(ca.getConstPool());
                b12.addInvokestatic(cc1, minf12.getName(), minf12
                                .getDescriptor());
                byte[] code12 = b12.get();
                ci.insert(index, code12);
                break;
        case Opcode.SWAP:
                MethodInfo minf13 = cf1.getMethod("Swap");
                Bytecode b13 = new Bytecode(ca.getConstPool());
                b13.addInvokestatic(cc1, minf13.getName(), minf13
                                .getDescriptor());
                byte[] code13 = b13.get();
                ci.insert(index, code13);
                break;
        case Opcode.POP:
                MethodInfo minf14 = cf1.getMethod("Pop");
                Bytecode b14 = new Bytecode(ca.getConstPool());
                b14.addInvokestatic(cc1, minf14.getName(), minf14
                                .getDescriptor());
                byte[] code14 = b14.get();
                ci.insert(index, code14);
```

```
                break;
            case Opcode.POP2:
                MethodInfo minf15 = cf1.getMethod("Pop2");
                Bytecode b15 = new Bytecode(ca.getConstPool());
                b15.addInvokestatic(cc1, minf15.getName(), minf15
                        .getDescriptor());
                byte[] code15 = b15.get();
                ci.insert(index, code15);
                break;
            case Opcode.NEWARRAY:
            case Opcode.ANEWARRAY:
                MethodInfo minf16 = cf1.getMethod("NewArray");
                Bytecode b16 = new Bytecode(ca.getConstPool());
                b16.addLdc("0NewArray");
                b16.addInvokestatic(cc1, minf16.getName(), minf16
                        .getDescriptor());
                byte[] code16 = b16.get();
                ci.insert(index, code16);
                break;
            case Opcode.IF_ACMPEQ:
            case Opcode.IF_ACMPNE:
            case Opcode.IF_ICMPEQ:
            case Opcode.IF_ICMPGE:
            case Opcode.IF_ICMPGT:
            case Opcode.IF_ICMPLE:
            case Opcode.IF_ICMPLT:
            case Opcode.IF_ICMPNE:
                IF_Condition++;
                MethodInfo minf17 = cf1.getMethod("ifcmp");
                Bytecode b17 = new Bytecode(ca.getConstPool());
                b17.addInvokestatic(cc1, minf17.getName(), minf17
                        .getDescriptor());
                byte[] code17 = b17.get();
                ci.insertEx(index, code17);
                break;
            case Opcode.IFEQ:
            case Opcode.IFGE:
            case Opcode.IFGT:
            case Opcode.IFLE:
            case Opcode.IFLT:
            case Opcode.IFNE:
```

```
                        case Opcode.IFNULL:
                        case Opcode.IFNONNULL:
                                IF_Condition++;
                                MethodInfo minf19 = cf1.getMethod("ifcond");
                                Bytecode b19 = new Bytecode(ca.getConstPool());
                                b19.addInvokestatic(cc1, minf19.getName(), minf19
                                                .getDescriptor());
                                byte[] code19 = b19.get();
                                ci.insertEx(index, code19);
                                break;
case Opcode.INVOKEVIRTUAL:
case Opcode.INVOKEINTERFACE:
case Opcode.INVOKESTATIC:
Native=0;
byte by1 = ca.getCode()[index + 1];
byte by2 = ca.getCode()[index + 2];
int indexx1 = (by1 << 8) | by2;
PrametersNumber = 0;
ReturnNumber = 0;
String className = null;
MethodName = null;
MethodrefType = null;
if (op == Opcode.INVOKEINTERFACE) {
        MethodName = cf.getConstPool()
        .getInterfaceMethodrefName(indexx1);
        MethodrefType = cf.getConstPool()
        .getInterfaceMethodrefType(indexx1);

        } else
        {
        className = cf.getConstPool().getMethodrefClassName(indexx1);

        if (className != null)
        {
        MethodName = cf.getConstPool().getMethodrefName(
        indexx1);
        MethodrefType = cf.getConstPool().getMethodrefType(indexx1);
        PrametersNumber = methodPrameters(MethodrefType);
        ReturnNumber = methodReturnValue(MethodrefType);
        cm.instrument(new ExprEditor() {
        public void edit(MethodCall m)
```

```
throws CannotCompileException {
String  MthodN = m.getMethodName();
if (MthodN.equalsIgnoreCase(MethodName)) {
try {
if (Modifier.isNative(m.getMethod()
.getModifiers())
|| (Modifier.isAbstract(m
.getMethod()
.getModifiers()))) {
PrametersNumber = methodPrameters(m.getSignature());
ReturnNumber = methodReturnValue(m.getSignature());
        Native=1;
        }
        } catch (NotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
                }
        }
}
});
}else
        {
        cm.instrument(new ExprEditor() {
        public void edit(MethodCall m)
        throws CannotCompileException {
        MethodName = m.getMethodName();
        try {
        if (Modifier.isNative(m.getMethod()
        .getModifiers())
        || (Modifier.isAbstract(m
        .getMethod()
        .getModifiers()))) {
        PrametersNumber = methodPrameters(m
        .getSignature());
        ReturnNumber = methodReturnValue(m
        .getSignature());
        Native=1;
        }
        } catch (NotFoundException e) {
        e.printStackTrace();
                                                }
```

```
                        }
                    });


                }
                if (op == Opcode.INVOKEVIRTUAL)
                    PrametersNumber++;
            }
            if (Native==0){
                MethodInfo minfMethod = cf1.getMethod("Method");
                Bytecode bby4 = new Bytecode(ca.getConstPool());
                //bby4.addIconst(InstrumentClass.run);
                bby4.addLdc(MethodName);
                bby4.addIconst(PrametersNumber);
                //bby4.addIconst(ReturnNumber);
                bby4.addInvokestatic(cc1, minfMethod.getName(),
                        minfMethod.getDescriptor());
                byte[] ccode = bby4.get();
                ci.insert(index, ccode);
                break;
            }
            else{
                if (MethodName.equalsIgnoreCase("write"))
                        {


                    MethodInfo minfWrite = cf1
                            .getMethod("NativeWrite");
                    Bytecode bby2 = new Bytecode(ca.getConstPool());
                    bby2.addIconst(PrametersNumber);
                    bby2.addIconst(ReturnNumber);
                    bby2.addInvokestatic(cc1, minfWrite.getName(),
                            minfWrite.getDescriptor());
                    byte[] codeee = bby2.get();
                    ci.insert(index, codeee);
                    temp = 1;
                    break;


                }
                else
            {
                MethodInfo minfN = cf1.getMethod("NativeMethod");
```

```
                        Bytecode bby3 = new Bytecode(ca.getConstPool());
                        //bby3.addIconst(InstrumentClass.run);
                        bby3.addIconst(PrametersNumber);
                        bby3.addIconst(ReturnNumber);
                        bby3.addInvokestatic(cc1, minfN.getName(), minfN
                                        .getDescriptor());
                        byte[] coode = bby3.get();
                        ci.insert(index, coode);
                        break;
                }
                }
            case Opcode.INVOKESPECIAL:
            byte bb6 = ca.getCode()[index + 1];
                    byte bb7 = ca.getCode()[index + 2];
                    int index67 = (bb6 << 8) | bb7;
                    String className1 = null;
                    if (op == Opcode.INVOKEINTERFACE) {
                            MethodN = cf.getConstPool()
                                            .getInterfaceMethodrefName(index67);
                            MethodrefType = cf.getConstPool()
                                            .getInterfaceMethodrefType(index67);
                            className1=cf.getConstPool().
                                 getInterfaceMethodrefClassName(index67);
                    }
                    else
        {
                            className1=cf.getConstPool().getMethodrefClassName(index67
                                 );
                    MethodrefType = cf.getConstPool().getMethodrefType(
                                    index67);
                    MethodN = cf.getConstPool().getMethodrefName(index67);
                    if (MethodN == null) {
                            cm.instrument(new ExprEditor() {
                                    public void edit(MethodCall m)
                                                    throws CannotCompileException {
                                            MethodN = m.getMethodName();
                                            PrametersNumber = methodPrameters(m
                                                            .getSignature());
                                            ReturnNumber = methodReturnValue(m
                                                            .getSignature());
```

```
                                        }
                                });
                        }
                }
                if (!MethodN.equalsIgnoreCase("<init>")) {
                        if (MethodN.equalsIgnoreCase("readBytes")) {
                                CtClass ct = pool.get(className1);
                                CtMethod ccm = ct.getDeclaredMethod(MethodN);
                                int modf = ccm.getModifiers();
                                if ((Modifier.isNative(modf))
                                                || (Modifier.isAbstract(modf))) {
                                        MethodInfo minfreadbyte = cf1.getMethod("
                                                NativeMethod");
                                        Bytecode bby1 = new Bytecode(ca.getConstPool());
                                        bby1.addIconst(methodPrameters(MethodrefType));
                                        bby1.addIconst(methodReturnValue(MethodrefType));
                                        bby1.addInvokestatic(cc1, minfreadbyte.getName(),
                                                        minfreadbyte.getDescriptor());
                                        byte[] codde = bby1.get();
                                        ci.insert(index, codde);
                                }
                        }

                else if(op==Opcode.INVOKESPECIAL)
                                {

                                        MethodInfo mfMethod = cf1.getMethod("Method");
                                        Bytecode be4 = new Bytecode(ca.getConstPool());
                                        be4.addLdc(MethodN);
                                        be4.addIconst(methodPrameters(MethodrefType)+1);
                                        be4.addInvokestatic(cc1, mfMethod.getName(),
                                                        mfMethod.getDescriptor());
                                        byte[] cde = be4.get();
                                        ci.insert(index, cde);
                        break;
                }
                                }else if (MethodN.equalsIgnoreCase("<init>")) {
                                        MethodInfo minf67 = cf1.getMethod("SpecialMethod");
                                        Bytecode b67 = new Bytecode(ca.getConstPool());
                                        b67.addInvokestatic(cc1, minf67.getName(), minf67
                                                        .getDescriptor());
```

```
                                byte[] code67 = b67.get();
                                ci.insert(index, code67);
                                break;
                        }
                case Opcode.MONITORENTER:
                case Opcode.MONITOREXIT:
                        MethodInfo minf373 = cf1.getMethod("Monitor");
                        Bytecode b373 = new Bytecode(ca.getConstPool());
                        b373.addInvokestatic(cc1, minf373.getName(), minf373 .getDescriptor
                                ());
                        byte[] code373 = b373.get();
                        ci.insert(index, code373);
                        break;
                case Opcode.IRETURN:
                case Opcode.LRETURN:
                case Opcode.FRETURN:
                case Opcode.DRETURN:
                case Opcode.ARETURN:
                        MethodInfo minf20 = cf1.getMethod("TReturn");
                        Bytecode b20 = new Bytecode(ca.getConstPool());
                        b20.addIconst(IF_Condition);
                        b20.addInvokestatic(cc1, minf20.getName(), minf20
                                        .getDescriptor());
                        byte[] code20 = b20.get();
                        ci.insert(index, code20);
                        break;
                case Opcode.RETURN:
                        MethodInfo minf23 = cf1.getMethod("Return");
                        Bytecode b23 = new Bytecode(ca.getConstPool());
                        b23.addIconst(IF_Condition);
                        b23.addInvokestatic(cc1, minf23.getName(), minf23
                                        .getDescriptor());
                        byte[] code23 = b23.get();
                        ci.insert(index, code23);
                        break;
                }
        }
        int i = 0;
        CodeIterator cii = ca.iterator();
        int ifoffsetAddress;
        int gotoffsetAddress;
```

```
while (cii.hasNext()) {
        i = cii.next();
        int op = cii.byteAt(i);
        if (op==Opcode.IFEQ ||
                        op==Opcode.IFGE ||
                        op==Opcode.IFGT ||
                        op==Opcode.IFLE ||
                        op==Opcode.IFLT ||
                        op==Opcode.IFNE ||
                        op==Opcode.IFNULL||
                        op==Opcode.IFNONNULL||
                        op==Opcode.IF_ACMPEQ ||
                        op==Opcode.IF_ACMPNE ||
                        op==Opcode.IF_ICMPEQ ||
                        op==Opcode.IF_ICMPGE ||
                        op==Opcode.IF_ICMPGT ||
                        op==Opcode.IF_ICMPLE ||
                        op==Opcode.IF_ICMPLT ||
                        op==Opcode.IF_ICMPNE ) {
                        byte bb1 = ca.getCode()[i + 1];
                        byte bb2 = ca.getCode()[i + 2];
                        int offset = (bb1 << 8) | (bb2);
                        ifoffsetAddress =i+offset;
                        if (i>ifoffsetAddress){
                                IF_Condition--;
                                MethodInfo minif2 = cf1.getMethod("Endif");
                                Bytecode b23 = new Bytecode(ca.getConstPool());
                                b23.addInvokestatic(cc1, minif2.getName(), minif2
                                                .getDescriptor());
                                byte[] code23 = b23.get();
                                cii.insert(i, code23);
                        }
                        if (i<ifoffsetAddress){
                                int opp = cii.byteAt(ifoffsetAddress -3);
                        if (opp!= Opcode.GOTO){
                                IF_Condition--;
                                MethodInfo minif2 = cf1.getMethod("Endif");
                                Bytecode b23 = new Bytecode(ca.getConstPool());
                                        b23.addInvokestatic(cc1, minif2.getName(),
                                             minif2
                                                .getDescriptor());
```

```
                                        byte[] code23 = b23.get();
                                        cii.insertEx(ifoffsetAddress, code23);
                                }
                                if (opp== Opcode.GOTO){
                                        IF_Condition--;
                                        byte bby1 = ca.getCode()[i+offset-2];
                                        byte bby2 = ca.getCode()[i+offset-1];
                                        int offs = (bby1 << 8) | (bby2);
                                        gotoffsetAddress=i+offset+offs-3;
                                        int insert= ifoffsetAddress-3;
                                        if (offs<0){
                                                MethodInfo minif1 = cf1.getMethod("Endif");
                                                        Bytecode b23 = new Bytecode(ca.
                                                             getConstPool());
                                                        b23.addInvokestatic(cc1, minif1.
                                                             getName(), minif1
                                                                     .getDescriptor());
                                                        byte[] code23 = b23.get();
                                                        cii.insertEx(insert, code23);
                                        }
                                        else{
                                        MethodInfo minif1 = cf1.getMethod("Endif");
                                        Bytecode b23 = new Bytecode(ca.getConstPool());
                                        b23.addInvokestatic(cc1, minif1.getName(), minif1
                                                        .getDescriptor());
                                        byte[] code23 = b23.get();
                                        cii.insert(gotoffsetAddress, code23);

                                        }
                                }
                                }
                        }



                }
                InstructionPrinter.print(cm, p);
                InstructionPrinter.print(cm, System.out);
        cc.writeFile();
                cc.defrost();
        } catch (Exception e) {
```

```
                    e.printStackTrace(System.out);
                    System.out.println("Error occured while redefining method "
                                    + cm.getLongName());
                }
        }
public static int methodPrameters(String s) {
                int PrametersNumber = 0;
                String ss;
                if (s!=null){
                if (s.indexOf("(") + 1 != (s.indexOf(")"))) {
                ss = s.substring(s.indexOf("(") + 1, s.indexOf(")"));
                        int i = 0;
                        while (i < ss.length()) {
                                if (Character.isLetter(ss.charAt(i))) {
                                        PrametersNumber++;
                                        if (ss.charAt(i) == 'L') {
                                                while (ss.charAt(i) != ';') {
                                                        i++;
                                                }
                                        }
                                        if (ss.charAt(i) == '[') {
                                                i++;
                                        }
                                        if (ss.charAt(i) == '(') {
                                                i++;
                                        }

                                }
                                i++;
                        }

                }
                }
                return PrametersNumber;
        }
public static int methodReturnValue(String s) {
                int ReturnNumber = 0;
                if (s!=null){
                String ss = s.substring(s.indexOf(")") + 1);
                int i = 0;
                while (i < ss.length()) {
```

```
                    if (Character.isLetter(ss.charAt(i))) {
                            ReturnNumber++;
                            if (ss.charAt(i) == 'L') {
                                    while (ss.charAt(i) != ';') {
                                            i++;
                                    }
                            }
                            if (ss.charAt(i) == '[') {
                                    i++;
                            }

                    }
                    i++;
            }
            if (ss.equals("V") == true) {
                    ReturnNumber = 0;
            }
            }
            return ReturnNumber;
        }
}
```

Listing C.5: Source code of EventRecognizer.java

```
package uk.ac.dmu.msarrab.vif.framework;
import java.lang.String;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Stack;
import javassist.NotFoundException;
/**
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class EventRecognizer {
        /**
         * Manipulates the Symbol tables, IFSs and IMFSs using stack and frame class
        */
        static Stack<HashSet<String>> IMFS = new Stack<HashSet<String>>();
        public static MyStack RuntimeFrame = new MyStack();
        public static void Store(Object ss, String s) {
```

```
            HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> st1 = new HashSet<String>();
            Iterator<String> ci = st.iterator();
            while (ci.hasNext()) {
                    Object e = ci.next();
                    st1.add(e.toString());
            }
            Object[] rray = st1.toArray();
            for (int i = 0; i < rray.length; i++) {
                    st1.add(rray[i].toString());
            }
            if (ss != null)
                    st1.add(ss.toString());
            RuntimeFrame.put(s, st1);
            StackPrint();
    }
    public static void iStore(int ss, String s) {
            HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> st1 = new HashSet<String>();
            Iterator<String> ci = st.iterator();
            while (ci.hasNext()) {
                    Object e = ci.next();
                    st1.add(e.toString());
            }
            Object[] rray = st1.toArray();
            for (int i = 0; i < rray.length; i++) {
                    st1.add(rray[i].toString());
            }
            st1.add(String.valueOf(ss));
            RuntimeFrame.put(s, st);
            StackPrint();
    }
    public static void astore() {
            RuntimeFrame.pop();
            RuntimeFrame.pop();
            RuntimeFrame.pop();
            StackPrint();
    }
    public static void Load(Object s, String ss) {
            HashSet<String> set = (HashSet<String>) RuntimeFrame.get(ss);
            set.add(s.toString());
```

```
        RuntimeFrame.push(set);
        StackPrint();
    }
    public static void Aload(Object s, String ss) {
        if (!RuntimeFrame.get(ss).equals(null)) {
            System.out.println("not null");

            HashSet<String> set = (HashSet<String>) RuntimeFrame.get(ss);
            set.add(s.toString());
            RuntimeFrame.push(set);
        } else {
            HashSet<String> set = new HashSet<String>();
            set.add(s.toString());
            RuntimeFrame.push(set);
        }
        StackPrint();
    }
    public static void Lload(Long s, String ss) {
        HashSet<String> set = (HashSet<String>) RuntimeFrame.get(ss);
        set.add(s.toString());
        RuntimeFrame.push(set);
        StackPrint();
    }

    public static void Fload(int run, Float s, String ss) {
        HashSet<String> set = (HashSet<String>) RuntimeFrame.get(ss);
        set.add(s.toString());
        RuntimeFrame.push(set);
        StackPrint();
    }
    public static void Iload(String ss) {
        System.out.println("ss= " + ss);
        HashSet<String> set = (HashSet<String>) RuntimeFrame.get(ss);
        set.add("Const");
        System.out.println("s= " + set);
        RuntimeFrame.push(set);
        StackPrint();
    }
    public static void Dload(Double s, String ss) {
        HashSet<String> set = (HashSet<String>) RuntimeFrame.get(ss);
        set.add(Double.toString(s));
```

```
        RuntimeFrame.push(set);
        StackPrint();
    }
    public static void LoadField(String s) {
        HashSet<String> st = new HashSet<String>();
        st.add(s.toString());
        RuntimeFrame.push(st);
        StackPrint();
    }
    public static void LoadIntField(int s) {
    HashSet<String> st = (HashSet<String>) RuntimeFrame.get(Integer
                        .toString(s));
        String aString = Integer.toString(s);
        st.add(aString);
        RuntimeFrame.push(st);
        StackPrint();
    }
    public static void StoreField(String s) {
        HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
        RuntimeFrame.put(s, st);
        StackPrint();
    }

    public static void Const() {
        HashSet<String> st = new HashSet<String>();
        st.add("Const");
        RuntimeFrame.push(st);
        StackPrint();
    }
    public static void tableswitch() {
        HashSet<String> st1 = (HashSet<String>) RuntimeFrame.pop();
        IMFS.push(st1);
        StackPrint();
    }
    public static void lookupswitch() {
        HashSet<String> st1 = (HashSet<String>) RuntimeFrame.pop();
        IMFS.push(st1);
        StackPrint();
    }

    public static void Dup() {
```

```
            HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
            RuntimeFrame.push(st);
            RuntimeFrame.push(st);
            StackPrint();
    }
    public static void Union() {
            HashSet<String> set1 = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> set2 = (HashSet<String>) RuntimeFrame.pop();
            set1.addAll(set2);
            RuntimeFrame.push(set1);
            StackPrint();
    }

    public static void New(Object s) {
            HashSet<String> st = new HashSet<String>();
            st.add(s.toString());
            RuntimeFrame.push(st);
            StackPrint();
    }
    public static void Return(int counterOfCondition) {
            while (counterOfCondition > 0) {
                    IMFS.pop();
                    counterOfCondition--;
            }
            RuntimeFrame.closeframe();
            StackPrint();
    }
    public static void TReturn(int counterOfConditions) {
            HashSet<String> ReturnValue = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> set = new HashSet<String>();
            while (!IMFS.empty()) {
                    HashSet<String> set1 = (HashSet<String>) IMFS.pop();
                    ReturnValue.addAll(set1);
                    set.addAll(set1);
            }
            Iterator<String> it = ReturnValue.iterator();
            while (it.hasNext()) {
                    String element = it.next();
                    HashSet<String> s = new HashSet<String>();
                    s.add(element);
                    IMFS.push(s);
```

```
            }
            while (counterOfConditions > 0) {
                IMFS.pop();
                counterOfConditions--;
            }
            StackPrint();
            RuntimeFrame.closeframe();
            RuntimeFrame.push(ReturnValue);
            StackPrint();
        }
        public static void Swap() {
            HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> st1 = (HashSet<String>) RuntimeFrame.pop();
            RuntimeFrame.push(st);
            RuntimeFrame.push(st1);
            StackPrint();
        }
        public static void ifcmp() {
            HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> st1 = (HashSet<String>) RuntimeFrame.pop();
            st.addAll(st1);
            System.out.println("1= " + st);
            System.out.println("2= " + st1);
            IMFS.push(st);
            StackPrint();
        }
        public static void ifcond() {
            HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
            IMFS.push(st);
            StackPrint();
        }
        public static void Endif() {
            IMFS.pop();
            StackPrint();
        }
        public static void Pop() {
            RuntimeFrame.pop();
            StackPrint();
        }
        public static void Pop2() {
            RuntimeFrame.pop();
```

```
            RuntimeFrame.pop();
            StackPrint();
    }
    public static void Method(String s, int j) {
            int f = 0;
            for (int i = 0; i < j; i++) {
                    HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
                    System.out.println("Start Method: " + s + " , pop()= " + st);
                    if (f == 0) {
                            RuntimeFrame.openframe();
                            f++;
                    }
                    RuntimeFrame.put(Integer.toString(i), st);
            }
            StackPrint();
    }
    public static void NativeMethod(int parameter, int returnValues) {
            HashSet<String> set1 = new HashSet<String>();
            for (int i = 0; i < parameter; i++) {
                    HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
                    set1.addAll(st);
            }
            for (int i = 0; i < returnValues; i++) {
                    RuntimeFrame.push(set1);
            }
            StackPrint();
    }
    public static void NewArray(String s) {
            HashSet<String> st = new HashSet<String>();
            st.add(s);
            RuntimeFrame.push(st);
            StackPrint();
    }
    public static void SpecialMethod() {
            HashSet<String> set1 = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> set2 = (HashSet<String>) RuntimeFrame.pop();
            HashSet<String> set3 = (HashSet<String>) RuntimeFrame.pop();
            set1.addAll(set2);
            set1.addAll(set3);
            RuntimeFrame.push(set1);
            StackPrint();
```

```
        }
        public static void NativeWrite(int Parameters, int rr) {
                HashSet<String> set1 = new HashSet<String>();
                for (int i = 0; i <= Parameters; i++) {
                        HashSet<String> st = (HashSet<String>) RuntimeFrame.pop();
                        set1.addAll(st);
                }
                System.out.println("Flow where? i do not know " + set1);
                for (int i = 0; i < rr; i++) {
                        RuntimeFrame.push(set1);
                }
        }

        public static void Monitor() {
                RuntimeFrame.pop();
                StackPrint();
        }
        public static void StackPrint() {
                System.out.println("Runtime Frames : "
                                + RuntimeFrame.currentframe().IFS);
                System.out.println("Runtime Frames : "
                                + RuntimeFrame.currentframe().Symbol_Table);
                System.out.println("Contens of IMFS: " + IMFS);
        }
}
```

Listing C.6: Source code of MyStack.java

```
package uk.ac.dmu.msarrab.vif.framework;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Set;
import java.util.Stack;
/**
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class MyStack {
        Stack<StackFrame> stack = new Stack<StackFrame>();
        public void openframe() {
```

```
        stack.push(new StackFrame());
    }
    public StackFrame closeframe() {
        return stack.pop();
    }
    public StackFrame currentframe() { return stack.peek(); }

    public void push(Set<String> element) {
        currentframe().IFS.push(element);
    }
    public Set<String> pop() {
        return currentframe().IFS.pop();
    }
    public Set<String> peek() {
        return currentframe().IFS.peek();
    }
        public HashSet<String> put(String key, HashSet<String> value) {
        return currentframe().Symbol_Table.put(key, value);
        }
        public HashSet<String> get(String key) {
            return currentframe().Symbol_Table.get(key);
        }
    public String toString() {
        return stack.toString();
    }
    class StackFrame {
        Hashtable<String, HashSet<String>> Symbol_Table = new Hashtable<String, HashSet<
            String>>();
        Stack<Set<String>> IFS = new Stack<Set<String>>();

        public HashSet<String> put(String key, HashSet<String> value) {
            System.out.println(key+" "+ value);
            return Symbol_Table.put(key, value);
        }
        public HashSet<String> get(String key) {
            return Symbol_Table.get(key);
        }
        public void clear() {
            IFS.clear();
        }
        public boolean empty() {
```

```
                return IFS.empty();
        }
        public Iterator<Set<String>> iterator() {
                return IFS.iterator();
        }
        public Set<String> lastElement() {
                return IFS.lastElement();
        }
        public Set<String> peek() {
                return IFS.peek();
        }
        public Set<String> pop() {
                return IFS.pop();
        }
        public Set<String> push(Set<String> item) {
                return IFS.push(item);
        }
        public int size() {
                return IFS.size();
        }
        public Object[] toArray() {
                return IFS.toArray();
        }
        public <T> T[] toArray(T[] a) {
                return IFS.toArray(a);
        }
        public String toString() {
                return IFS.toString() + Symbol_Table.toString();
        }
    }
}
```

Listing C.7: Source code of RunTimeChecker.java

```
package uk.ac.dmu.msarrab.vif.framework;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.lang.String;
import java.util.ArrayList;
import java.util.Iterator;
```

```
import java.util.List;
import java.util.Set;
/**
 * @author Mohamed Sarrab (STRL, DMU, UK)
 * Msarrab@dmu.ac.uk
 */
public class RunTimeChecker {
	/**
	 * Checks the flow of the information against the information flow policy
	 * and monitor the flow to the user
	*/
	public static void Check(Set<String> st1) throws IOException {
		List<String> list = new ArrayList<String>(st1);
		for (int j = 0; j < list.size(); j++) {
			String value = list.get(j);
			if (!value.startsWith("java.") && (!value.startsWith("["))
					&& (!value.startsWith("0")) && (!value.startsWith("1"))
					&& (!value.isEmpty())) {
				System.err.println(value + " will flow to -—> System.out");
				BufferedReader stdIn = new BufferedReader(
						new InputStreamReader(System.in));
				stdIn.readLine();
				break;
			}
		}
	}
	public static void AskUser(Set<?> st) {
		System.out.println("Checked Set= " + st);
		Iterator<?> rr = st.iterator();
		while (rr.hasNext()) {
			Object element = rr.next();
			System.out.println("==========" + element.toString());
		}
	}
}
```

Appendix D

Byte Code of Case Studies 1 and 2

Due to the big size of the instrumented bytecode of the class files that are used in case studies 1 and 2, the thesis has an associated CD that has the original and instrumented bytecode of both case studies (Section 8.2 and Section 8.3).